KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Composition Pattern for Constraint-based Programming

with application to force-sensorless robot tasks

Dominick Vanthienen

Supervisor:
Prof. dr. ir. Joris De Schutter
Prof. dr. ir. Herman Bruyninckx, co-supervisor

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering Science

January 2015

Composition Pattern for Constraint-based Programming

with application to
force-sensorless robot tasks

Dominick VANTHIENEN

Examination committee:
Prof. dr. ir. Jean Berlamont, chair
Prof. dr. ir. Joris De Schutter, supervisor
Prof. dr. ir. Herman Bruyninckx, co-supervisor
Prof. dr. ir. Jan Swevers
Prof. dr. ir. Geert Deconinck
Prof. dr. ir. Tinne De Laet
Prof. dr. Tom Holvoet
Prof. dr. Rachid Alami
(LAAS/CNRS Toulouse)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering Science

January 2015

Uitgegeven in eigen beheer, Dominick Vanthienen, Celestijnenlaan 300B box 2420, B-3001 Heverlee (Belgium)

ISBN 978-94-6018-931-9

D/2015/7515/1

Preface

Juni 2008, diploma op zak en contract al lang getekend, *'Ik ga nooit doctoreren'*... November 2009 –een jaartje wijzer– sta ik terug in Leuven en waag ik me dan toch aan dat doctoraat. De start van vijf leuke, avontuurlijke, soms ook uitputtende, maar vooral leerrijke jaren.

Tijdens deze jaren heb ik me kunnen verdiepen in de breedte en diepte van de robotica in zijn vele facetten: mechanica, controle, software... Bovendien hebben de jaren me een nieuwe kijk op de wereld gegeven, letterlijk en figuurlijk, via internationale conferenties, *summer schools*, projecten... Het doctoraat heeft me ook tot aan de *bleeding edge* van de technologie gebracht, wat toch de kwaliteit van onze onderzoeksgroep robotica onderschrijft: we deden onderzoek met *drones* jaren voor Amazon aankondigde het leveren van pakketjes met drones te onderzoeken; ik was erbij toen de eerste PR2s 'afstudeerden' en naar de grote, bekende onderzoeksgroepen in de wereld gingen (MIT, Stanford ... en Leuven); etc.

Bedankt prof. Joris De Schutter en prof. Herman Bruyninckx, promotoren van mijn doctoraat, om me deze kans te geven, om in mij te geloven en om me de vrijheid in onderzoek te geven. (Wat was die oorspronkelijke titel ook weer? Manipulatie met quadrotors?). Ook wil ik jullie bedanken voor de wijze raad en de inzichten die jullie me gegeven hebben. Joris, bedankt voor je geduld, de constructieve feedback en de vele taaltips Engels. Herman –*goeroe*– bedankt voor je visie en de vele discussies, waar we soms 'het al eens waren, maar het nog niet beseften'. Daarnaast zou ik ook de jury willen bedanken voor het nalezen van mijn doctoraatstekst en voor de tips ter verbetering van de tekst. Prof. Alami, je vous remercie pour vos commentaires constructifs sur ma thèse et ma recherche. Bedankt prof. Berlamont om de jury voor te zitten. Verder zou ik ook prof. Reynaerts willen bedanken om deel uit te maken van mijn begeleidingscommissie.

Ik zou ook het FWO willen bedanken voor het financieren van het project waardoor dit werk tot stand is kunnen komen.

I would like to thank my colleagues, especially those with whom I shared the office. They are my peers, *mes compagnons de route*, my friends. Together we had good and bad times, sharing laughs and tears; together we shared beers after office hours, but also code and deadline worries. Thank you, you made it a pleasure to come to the office!

In het bijzonder wil ik Tinne bedanken, directe buur op bureau tijdens de eerste jaren en grootste slachtoffer van mijn oneindig aantal vragen en mijn neiging om ideeën mondeling af te toetsen. Tinne, bedankt ook voor het vele nalezen van mijn publicaties én thesistekst, bedankt voor de wafels op de 'wafelenbak' en bedankt om –hoogzwanger– te helpen bij het opnemen van de comanipulatie-experimenten.

Verder is er Wilm, altijd beschikbaar voor een discussie of openhartig babbeltje over om het even welk onderwerp. Thank you Enrico, Enea, Niccolo –our *Italian gang*– for the many dinners we had and the upgrade of our coffee standards. Dan is er Steven, de andere helft van de *quadrotor-guys* van het eerste uur, met wie er altijd een leuke sfeer heerste.

Further I would like to thank Marcus –eternal *zen* in all our group's notoriously fierce discussions– for introducing Lua and uplifting our code to the *(meta)-meta-level*. Thank you Lin, for the many tips on China and the plentiful cookies you brought us after each trip to China!

Bedankt ook Hans en Koen –*hands-on guys*– altijd enthousiast voor een 'projectje'. Verder ook bedankt Bart en Jon –de huidige *quadrotor-guys*– voor de leuke momenten met de drones. Thank you Nico for the delicious Japanese dinners you organized for us! Bedankt ook Ruben en Peter om me te helpen om die eerste *segfaults* te debuggen. Thank you Sebastian and Azamat for the many talks and for being my German PO box. Dan zijn er –de *not-so-new kids on the block*– Jonas, Maxim en Kevin. Bedankt om mijn code uit te proberen en voor de sfeer in de groep nieuw leven in te blazen.

Furthermore, thank you Sergio, Gianni, Simona and the whole Italo-Spanish community of our department for the mediterranean dinners and thank you for reviving the social activities at our department.

I will treasure the good memories with you, my colleagues: the trip to Alaska with Mauro, where we did some bear-spray experiments; the trip to San Francisco with the unexpected lesson in anthropology (and anatomy) with Jon, Steven, Markus and Wilm; the trip to Yosemite in our 'Crown Vic' with Markus and Wilm; the robot restaurant in Tokyo with Enrico, Enea and Johan; the wonderful

location and food of the BRICS research camp at Bertinoro with Enea, Steven and Herman; the visits to Willow Garage, Stanford and Berkeley (thank you Joachim for having me stay at your place) with Koen and Eric (sorry Koen for the 'small' detour to get to Willow Garage); and the numerous long days and nights coding wherever in the world (Bremen, Munich, Freiburg, the Bay Area, etc.) ...

Furthermore, I would like to thank Rubens, the PR2 of our group. Like a groupie of a 'real' star, I could stand with you in the spotlight, but had to endure the troubles behind the scene. Many hours we have spent together, during which I discovered –and hopefully changed for the better– your inner self.

Verder zijn er ook de ATP-ers Bertram, Jean-Pierre, Karin, Lieve, Carine, Regine, José, Marijke, Anja, Marina, Peter, Valérie, Eddy, Jo, Pascal, Gunther, Jan en Ronny, die ik zou willen bedanken voor hun ondersteuning bij het doorworstelen van praktische en administratieve problemen.

Bedankt aan alle thesisstudenten die met hun inzet hebben kunnen bijdragen aan deze thesis. Ook bedankt aan de medeorganisatoren van de *Soirée pratiques* om onze studenten tot echte *makers* te maken en voor het helpen organiseren van de sumorobotcompetitie.

Ook wil ik verder al mijn familie en vrienden bedanken voor hun steun doorheen de jaren. Bedankt Frederik –mede nieuw-Leuvenaar en lotgenoot– en David voor de morele ondersteuning tijdens de afgelopen jaren. Bedankt Lynn, Ben, mama en papa voor de warme thuis waar ik altijd terecht kan en de kans en het geloof in mij om dit Leuvens avontuur tot een goed einde te brengen. Ook bedankt moeke en vake, bomma en bompa, voor de goede zorgen en steun doorheen de jaren.

Bedankt mijn allerliefste Carolien, *PhD widow* voor te lange tijd. Bedankt voor je liefde en om me altijd te steunen, ondanks alle weekends, avonden en feestdagen die deze tekst opeiste.

Ik zou mezelf niet zijn, mocht deze tekst niet geschreven zijn vanuit het hart, vlak voor het ter perse gaan. Daarom bedankt aan iedereen die ik nu vergeet maar ongetwijfeld bijgedragen heeft tot dit succes.

Dominick Vanthienen,
Heverlee, januari 2015.

Abstract

Robot platforms and applications are becoming more autonomous and complex. Many of these robots, and in particular service robots, evolve from a highly structured, human-shielded environments to a less structured, human-populated environments. Moreover, these robots are increasingly expected to interact physically and cognitively with humans and their environment.

The overall complexity of these robot systems requires the integration of knowledge of many domains and from multiple experts. Dealing with this integration challenge requires a systematic approach and knowledge-driven, flexible, reusable, and adaptable software.

This dissertation makes two complementary contributions: firstly, it provides a systematic approach to deal with the complexity of robot systems in general; and secondly, it provides a way to integrate force-sensorless wrench control in service robots.

As its first contribution, this dissertation introduces the Composition Pattern as a uniform and easy-to-grasp systematic approach to deal with complexity, from software architecture to behavior composition. The Composition Pattern defines a Composite Functional Entity as first-class citizen of system design. This Composite Functional Entity separates Coordinator, Composer, Scheduler, Configurator(s), Monitor(s), Communicator(s) and Functional Entities. Each Functional Entity can be in itself a Composite Functional Entity, hence can lead to a hierarchy of interacting entities. The Composition Pattern builds on the metamodeling concept, which considers all entities to be models, not objects. The former forms a constructive way to implement the 5Cs approach of separation of concerns; the latter separates concepts of the domain from implementation details, which makes the models more portable to other robot platforms.

Concretely, the Composition Pattern is applied as an architectural pattern to refactor the iTaSC constraint-based programming software framework. This

resulted in a framework of more reusable, flexible, robust, and adaptable entities. Furthermore, the Composition Pattern is used to structure and formalize constraint-based programming in a domain-specific language (DSL). This DSL enables developers and users to easier (and hence faster) understand and (re)program constraint-based programming applications, since (i) it provides a template of the system, (ii) it enables automatic model verification, and (iii) it enables manual or automatic code generation to the software framework ecosystem of choice.

As its second contribution, this dissertation presents a novel force-sensorless wrench control scheme for velocity controlled robots. This control scheme does not require a precise dynamic model of the robot, environment, or contact point; nor does it require an expensive and complex force sensor. Furthermore, it allows the combination of the wrench control constraints with other constraints, and it features a reference adaptation factor, which can be applied to impose a desired transient behavior on the applied wrench. Experimental validation of the control scheme, through the integration in the resolved-velocity iTaSC approach and software framework, proves that a stable, constant contact wrench can be reached in a repeatable way, and with an accuracy that fits service robot tasks.

In addition, this dissertation integrates both contributions in a force-sensorless and bimanual human-robot comanipulation application. The application shows complex behavior emerging from the composition of constraints to instantaneously and concurrently (i) keep the robot grippers parallel, (ii) null wrenches applied to the grippers, (iii) avoid obstacles, (iv) maintain visual contact with a person, (v) avoid joint limits, and (vi) optimize joint configuration. These constraints are expressed in multiple and different generalized task spaces.

The software implementation of the DSL, the iTaSC software framework, the wrench control schemes, and the comanipulation application are all made publicly availabe under an open-source license.

Beknopte samenvatting

Robotplatformen en -applicaties worden steeds autonomer en complexer. De omgeving waarin deze robots –en in het bijzonder hulprobots (*service robots*)– opereren, evolueert van een sterk gestructureerde, van de mens afgezonderde omgeving naar een minder gestructureerde, met de mens gedeelde omgeving. Bovendien vragen deze nieuwe toepassingen dat robots fysiek en cognitief interageren met mensen en hun omgeving.

De globale complexiteit van deze robot systemen vereist de kennisintegratie van verschillende domeinen en experts. Het succesvol benaderen van deze integratie vereist een systematische aanpak en kennisgedreven, flexibele, herbruikbare en aanpasbare software.

Dit proefschrift levert twee complementaire bijdragen. Ten eerste biedt het een systematische aanpak om met de geschetste complexiteit van robotsystemen om te gaan. Ten tweede biedt het een manier om krachtsensorloze krachtcontrole met hulprobots uit te voeren.

Als eerst bijdrage introduceert dit proefschrift het Compositiepatroon als uniforme en eenvoudig begrijpbare aanpak om met complexiteit om te gaan, van software-architectuur tot het samenstellen van robotgedrag. Het Compositiepatroon definieert een Composiet Functionele Entiteit als 'eerste klasse entiteit' (*first-class citizen*) van systeemontwerp. Deze Composiet Functionele Entiteit (*Composite Functional Entity*) bestaat uit een Coördinator, Samensteller (*Composer*), Activiteitplanner (*Scheduler*), Configurator(en), Monitor(s), Communicator(en) en Functionele Entiteiten (*Functional Entities*). Elk van de Functionele Entiteiten kan op zich een Composiet Functionele Entiteit zijn, om zo een hiërarchie van interagerende entiteiten te vormen. Het Compositiepatroon bouwt verder op het concept van meta-modelleren (*meta-modeling*). Meta-modelleren beschouwt alle entiteiten als modellen in tegenstelling tot de gangbare praktijk om entiteiten als objecten te beschouwen. Het Compositiepatroon biedt een constructieve manier om de

informatiebekommernissen te scheiden (*separation of concerns*) volgens de 5C-aanpak. Deze 5C-aanpak scheidt concepten uit het toepassingsdomein van implementatiedetails, wat de modellen meer overdraagbaar maakt naar andere robotplatformen.

Meer concreet past dit proefschrift het Compositiepatroon als architecturaal patroon toe op de herwerking van het iTaSC software-raamwerk voor beperkingsgebaseerd programmeren. Dit raamwerk bevat meer herbruikbare, flexibele, robuuste en aanpasbare entiteiten. Bovendien gebruikt dit proefschrift het Compositiepatroon om beperkingsgebaseerd programmeren te structureren en formaliseren. Hiertoe definieert het proefschrift een domeinspecifieke taal (*domain-specific language*, DSL). Deze DSL laat ontwikkelaars en gebruikers toe om eenvoudiger –en dus sneller– toepassingen die gebruik maken van beperkingsgebaseerd programmeren aan te passen en te begrijpen. Deze eenvoud resulteert uit (i) het gebruik van de DSL als sjabloon voor een toepassing, (ii) de automatische verificatie van het model en (iii) de mogelijkheid die de DSL biedt om (manueel of automatisch) code te genereren voor het software-raamwerk naar keuze.

Als tweede bijdrage introduceert dit proefschrift een nieuw krachtsensorloos kracht- en momentcontroleschema voor snelheidsgestuurde robots. Dit schema vereist geen nauwkeurig dynamisch model van de robot, de omgeving of het contact tussen beide, noch vereist het een dure en complexe krachtsensor. Bovendien laat het controlschema toe om kracht- en momentcontrole te combineren met andere taken en bevat het een factor om een gewenst transiënt gedrag op te leggen bij het uitoefenen van krachten en momenten. Experimentele validatie van het controleschema, door zijn integratie in de snelheidsopgeloste iTaSC-methodologie en -software-raamwerk, bewijst dat een stabiele en constante contactkracht kan worden uitgeoefend op een herhaalbare manier en met een nauwkeurigheid die volstaat voor hulprobottaken.

Daarnaast integreert dit proefschrift beide bijdragen in een krachtsensorloze en bimanuele mens-robot comanipulatietoepassing. De toepassing toont hoe de samenstelling van beperkingen om –ogenblikkelijk en gelijktijdig– (i) de robotgrijpers parallel te houden, (ii) de krachten en momenten uitgeoefend op de grijpers te compenseren, (iii) obstakels te vermijden, (iv) visueel contact met een persoon te houden, (v) gewrichtslimieten te vermijden en (vi) een optimale robotconfiguratie te behouden, resulteert in complex robotgedrag. Deze beperkingen zijn uitgedrukt in verschillende veralgemeende taakruimten.

De software-implementatie van de DSL, het iTaSC-software-raamwerk, de controleschema's, en de comanipulatietoepassing zijn publiek beschikbaar gemaakt onder een openbronlicentie.

Symbols, Abbreviations and Definitions

General Rules on Symbols

MATRIX	bold, capital letter, e.g. $\boldsymbol{A}$
SCALAR	letter, e.g. a
VECTOR	bold letter, e.g. $\boldsymbol{u}$

General Rules on Text Fonts

italic	Italic font indicates *models*.
TELETYPE FONT	Teletype font indicates `meta-models`.

General Rules on sub- and superscripts

$\bullet^{\#}$	*Moore-Penrose Pseudo-inverse*
$\bullet_d$	*desired*
$\bullet_W$	*weighted*
$\bullet_{(m \times n)}$	*matrix of m rows and n columns*
$\ddot{\bullet}$	*second time derivative*
$\dot{\bullet}$	*first time derivative*
$\hat{\bullet}$	*estimate*

Abbreviations

AI	Artificial Intelligence

CBP	Constraint-Based Programming
CC	Constraint-Controller
CO	Constraint-Output
DOF	Degrees-Of-Freedom
DSL	Domain-Specific Language
EMF	Eclipse Modeling Framework
FSM	Finite-State Machine
iTaSC	instantaneous Task Specification using Constraints
LOC	Lines Of Code
LWR	Light-Weight Robot (e.g. KUKA LWR)
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MIT	Massachusetts Institute of Technology
OMG	Object Management Group
OOP	Object Oriented Programming
OROCOS	Open Robot Control Software
PID	Proportional-integral-derivative Controller
ROS	Robot Operating System
RTT	Real Time Toolkit of OROCOS
SG	Setpoint Generator
TSR	Task Space Representation
UAV	Unmanned Aerial Vehicle
uMF	Micro Modeling Framework
VKC	Virtual Kinematic Chain
XML	Extensible Markup Language
rFSM	reduced Finite State Machine

Constraint-Based Programming and Control Related Symbols

$\boldsymbol{A}$	Augmented Jacobian, for a single task space $\boldsymbol{C_q} - \boldsymbol{C_f}\boldsymbol{J_f}^{-1}\boldsymbol{J_q}$
$\{b\}$	Base frame, the reference frame of a robot.
$\boldsymbol{\omega}$	Angular velocity vector, in 6D space expressed as three coordinates in a right-handed, Cartesian coordinate system
$\boldsymbol{\tau}$	Torque vector, in 6D space expressed as three coordinates in a right-handed, Cartesian coordinate system
$\boldsymbol{\tau}_{ext}$	External torque vector, torque applied by the robot joints when in robot-environment contact
$\boldsymbol{f}$	Force vector, in 6D space expressed as three coordinates in a right-handed, Cartesian coordinate system
$\boldsymbol{u}$	Control input to the robot joints (desired torques)
$\boldsymbol{v}$	Linear or translational velocity vector, in 6D space expressed as three coordinates in a right-handed, Cartesian coordinate system
$\boldsymbol{C_f}$	Feature coordinate selection matrix
$\boldsymbol{\chi_f}$	Feature coordinates, auxiliary coordinates that define the feature space (generalized task space) between two object frames.
$\boldsymbol{C_q}$	Joint coordinate selection matrix
$\boldsymbol{C}_r$	Robot joint damping matrix
$\boldsymbol{e}_{\dot{\boldsymbol{q}}}$	Robot joint velocity error
$\{f\}$	Feature frame, auxiliary frame defined on descriptive features of the generalized task space, e.g. the point of a pencil.
$\boldsymbol{FF}$	Feed-forward multiplication factor
$\boldsymbol{I}$	Unity matrix
$\boldsymbol{I}_r$	Inertia matrix of the robot joints
$\boldsymbol{J_q}$	Robot jacobian
K_a	Reference adaptation factor (scalar)
$\boldsymbol{K}_a$	Reference adaptation factor (matrix)
$\boldsymbol{K}_0$	Robot-environment stiffness matrix

$\boldsymbol{K}_v$	Proportional gains of the robot joint velocity controllers
n_c	Number of constraints
$n_{c,sel}$	Number of selected constraints
n_q	Number of controllable robot joints
$\{o\}$	Object frame, a frame defined on a robot or object, where a task will take effect.
$\boldsymbol{P}_{N(i)}$	Projection in the null space of the ith Jacobian
$\dot{\boldsymbol{q}}_{d1..i}$	Desired joint velocity as a result from all constraints from the first i priority levels
$\boldsymbol{q}$	Robot joint coordinates
$\dot{\boldsymbol{q}}_d$	Desired joint velocities, output of the optimization problem in a resolved-velocity formulation.
$\boldsymbol{S}$	6 by n_{sel} selection matrix. Each column and row contains one 1, other values are 0.
$\mathbf{t}$	$= \begin{bmatrix} \boldsymbol{v} \\ \boldsymbol{\omega} \end{bmatrix}$ Twist: velocity of a rigid body consisting of an angular velocity $\boldsymbol{\omega}$ around a screw axis and a linear velocity $\boldsymbol{v}$ along this axis.
$\boldsymbol{w}$	$= \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{\tau} \end{bmatrix}$ Wrench: forces and torques applied on a rigid body in a certain point
$\boldsymbol{w}_c$	Wrench applied by a robot on the environment or other body
$\boldsymbol{w}_d$	Desired wrench
$\boldsymbol{w}_{d,sel}$	Part of a desired wrench in selected directions $\boldsymbol{S}_c$
$\dot{\boldsymbol{y}}_d^\circ$	Desired derivatives of the output values after control, input to the optimization problem.
$\boldsymbol{y}$	Output coordinates, function of the robot joint coordinates and feature coordinates.
$\dot{\boldsymbol{y}}_{ex}$	Expected output coordinates, setpoint for the output coordinates
F_d	Desired force
n_q	Number of robot joints
n_{sel}	Number of selected cartesian directions to apply force control

Constraint-Based Programming Related Definitions

CONSTRAINT-BASED PROGRAMMING A general approach that states a problem as a set of (possibly partial or uncertain) conditions (constraints) that need to be satisfied, from which a computer program deduces a solution (solving).

CONSTRAINT-BASED TASK SPECIFICATION is an application of constraint-based programming on the robotics domain, where the overall robot task is defined as a composition of individual composable constraints. The robot solves the resulting constrained optimization problem for a behavior that satisfies the constraints.

CONSTRAINT Condition or property that needs to be satisfied.

FEATURE SPACE Space defined between two object frames and characterized by a set of feature coordinates. A task imposes a set of constraints on a feature space.

GENERALIZED TASK SPACE Task space, whether defined as a configuration, sensor, manipulation, constraint, task or feature space.

ITASC APPLICATION A concrete application in the iTaSC framework, filled in the reference architecture

ITASC FRAMEWORK An instantiation of the iTaSC Reference Architecture building on the Orocos framework and rFSM. It provides the tools to deploy and manage an iTaSC application.

ITASC WORKFLOW The procedure to create a constraint-based program following the rules of iTaSC

ITASC/CBP REFERENCE ARCHITECTURE The definition of the entity types (meta-models) that compose a constraint-based programming application and defines the interaction between these entity types (meta-models). However, it does not state how to fulfill or implement the functionality.

Software Development Related Definitions

(SOFTWARE) ARCHITECTURAL PATTERN *"An* **architectural pattern** *is a named collection of architectural design decisions that are applicable to a recurring design problem, parameterized to account for different software development contexts in which that problem appears. [...] It's a tactical design tool."* (Taylor et al. [152])

(Software) Architectural Style *"An* **architectural style** *is a named collection of architectural design decisions that are applicable in a given development context, constrain architectural design decisions that are specific to a particular system within that context, and elicit beneficial qualities in each resulting system. [...] It's a strategic design tool."* (Taylor et al. [152])

(Software) Architecture *"A software system's* **architecture** *is the set of principal design decisions made about the system. It is the blueprint for a software system's construction and evolution."* (Taylor et al. [152])

(Software) Reference Architecture *"A* **reference architecture** *is the set of principal design decisions that are simultaneously applicable to multiple related systems, typically within an application domain, with explicitly defined points of variation."* (Taylor et al. [152])

Composite Functional Entity A Composite Functional Entity composes other Functional Entities and Support Entities In the Composition Pattern, each Functional Entity can be replaced by a Composite Functional Entity.

Design Decision A **design decision** constitutes the choices made by a developer relating to system structure, functional behavior, interaction, non-functional properties, or implementation. Whether a design decision is a *principal* design decision depends on the goals and stakeholders [152].

Entity This thesis considers an **entity** a concept or model that maps to software components, agents, objects, modules, processes, activities...

Functional Entity A Functional Entity conforms to algorithms or computations (data processing).

Implementation An **implementation** is a piece of software, it is an *instance of*, or *represented by* a model. From a model a concrete implementation can be generated or hand-coded.

Meta-Meta-Model A **meta-meta-model** presents a language to describe a meta-model. A meta-model *conforms to* a meta-meta-model, a meta-meta-model possibly *conforms to* another meta-meta-model.

Meta-Model A **meta-model** presents a language to describe a model. A model *conforms to* a meta-model, a meta-model *conforms to* a meta-meta-model.

MODEL A **model** captures a view or aspect of a system, it groups semantics. A model *conforms to* one or more meta-models.

SOFTWARE FRAMEWORK ECOSYSTEM **Software framework ecosystems** provide a software development environment and the tooling to implement a certain architecture or application. The *ecosystem* addition refers to the fact that these frameworks form the centre of more specialized framework development or integration efforts.

SUPPORT ENTITY Support Entities 'manage' functional entities, by handling configuration, composition, coordination, monitoring, and scheduling.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A typical workday evening, after an exhaustive day at work, your autonomous car drives you home. You're barely out of the car and there is Jeeves, your robot butler. It greets you, puts your slippers on your feet, and hands you a whiskey while you make yourself comfortable in the couch. You call James, the robot cook, and order it to prepare dinner and set the table.

This situation could be the opening of a science-fiction book, however forms the topic of current research projects. For example, the DARPA Grand Challenges [38, 40] and its successor the DARPA Urban Challenge [39] formed, already in 2004, the foundation of the autonomous cars under development by almost all major car manufacturers. Research projects such as the RoboHow project [133] aim at developing autonomous service robots that can perform complex everyday human-scale activities in interaction with humans and the environment. These activities are not performed in a conditioned environment (robot-in-a-cage) but in a human-populated environment, such as at home or at the office. Moreover, the robot has cognitive capabilities to acquire new skills from the web [158] or by observing humans.

These examples show the variety and extent of the challenges in robotics *today*. This dissertation contributes to the solution of some of these challenges.

1.1 Motivation

The examples above demonstrate the growing *complexity* of robot platforms and applications. A robotic system has evolved from a pre-programmed or

teleoperated manipulator to a plethora of ever more autonomous systems. These systems do not necessarily consist of a single platform, but can consist of different cooperating agents. Each of these agents forms on its own a composition of different hard- and software subsystems, ranging from laser scanners to robot arms on a mobile base.

More of these robot systems move out of their safety cage and *interact physically and cognitively with humans*. Moreover, human users expect robots to be increasingly able to reason about their own functionalities and structure, and to explain behavior. To this end, the *perception and cognitive capabilities* of the robot platforms have to increase exponentially.

One of the market segments of complex robot systems that is expected to grow significantly is *service robotics*. These robots operate in unstructured environments, i.e. the robot has no prior knowledge on the location and characteristics of everything around it. However, a service robot has to interact with this unstructured environment, where it has to *apply forces and torques*[*] *on this environment*. In this application it is unrealistic nor necessary to obtain a full dynamic model of the environment. Moreover, integration of force-torque sensors comes at a cost. Avoiding this cost is favorable for mass production of service robots.

Developing robot systems as outlined above requires the *integration of the knowledge of many domains*, even beyond the traditional engineering domains. Moreover, due to its scale, this development is not a single person's job, but the result of the cooperation of *teams of developers*, each with a different domain of expertise.

The overall complexity of robot systems development demonstrates the *need for knowledge-driven, flexible, reusable, and adaptable software* to integrate the different subsystems and knowledge. Moreover, robot systems development does not limit itself to a single –although integrated– system: the different entities that form the system should be *replaceable and reusable*. For the software aspects, there should be support for easy reprogramming (*code refactoring*), the reuse of legacy code, and platform portability. (Still today hard- and software platform specific code is far too common.) Furthermore, the resulting integrated system should still be verifiable and debuggable.

This dissertation has two complementary goals. The first goal is to provide a systematic way to deal with the presented complexity of robot systems in general, and the second goal is to provide a way to integrate force-sensorless wrench control in service robots.

[*]The force-torque vector characterizing such an interaction is defined as a *wrench*.

1.2 Objectives

The following research objectives aim at realizing the two abovementioned goals.

The first objective is to **develop a systematic approach to create robot applications that are more flexible, robust, reusable, and adaptable.** As outlined in the motivation, the opposing trends of increasing scale and complexity of robots and their applications versus the demand for higher versatility of these applications, form a major challenge in need for a systematic approach.

The second objective is to **apply this approach to create applications based on constraint-based programming [46, 137], a constrained optimization approach to robot task specification and control.** The research group has extensive experience with constraint-based programming and encountered the deficiencies of former approaches to create robot applications. Moreover, the interplay of task specification, execution and monitoring forms a primary example of a robotics system involving 'planning', 'sensing', 'control' and 'world modeling' functionalities.

The third and fourth objective relate to this second objective. More concretely, the third objective is to **develop a domain specific language for constraint-based programming that supports the systematic approach of the first objective and incorporates its benefits,** and the fourth objective is to **develop a flexible, robust, and reusable software framework for constraint-based programming.**

The fifth objective relates to the second goal mentioned in the motivation. Many robot applications involve situations where contact between the robot and the environment needs to be made and forces applied. However, in many of these situations the required precision on the applied force is low, or the available robot platform is unable to perform wrench control tasks with high precision, for example because of safety constraints. This objective is to **develop a force-sensorless wrench control scheme for velocity controlled (service) robots, which does not require a precise dynamic model of the robot, environment, or contact point, but offers sufficient wrench setpoint regulation precision. Moreover, it should allow the combination of the wrench control constraints with other constraints.**

The sixth objective is to **integrate and validate the results achieved for the other objectives in a service robot application in which a PR2 robot [173] has to help a person in carrying an object, such as a table or a plate, as shown in Figure 1.1.**

Figure 1.1: Force-sensorless human-robot comanipulation in a restaurant. A robot helps a human carrying a plate while avoiding obstacles, maintaining visual contact with the operator, and avoiding unnatural poses. Photo by KU Leuven - Rob Stevens, published with permission.

The PR2 robot consists of two robot arms on a mobile platform. Moreover, it has extra degrees-of-freedom (DOF) in its head and back, amounting to a total of twenty DOF that have to be controlled.

This redundant robot has to

- comanipulate an object with a person,
- avoid dynamic and static obstacles,
- maintain visual contact with the person and the manipulated object, and
- avoid unnatural poses.

In order to comanipulate the object, the robot has to sense the forces and torques (i.e. the *wrench*) applied on the object by the human. Moreover, it

has to react to it and provide a following behavior. The PR2 platform does not have force sensors, therefore a force-sensorless wrench control technique is needed.

1.3 Approach

The approach of this dissertation to deal with the complexity described above relies on four important concepts: constrained optimization, separation of concerns, metamodeling, and hierarchy. The following sections introduce these concepts, an in-depth discussion can be found in chapter 2. The last section discusses the approach to force-sensorless wrench control.

1.3.1 Constrained optimization

Constrained optimization pertains to the (mathematical) *optimization* of an objective function with respect to a number of variables. These variables are subject to *constraints* that need to be taken into account when solving the optimization problem. This *objective function* is a cost or energy function that needs to be minimized, or a utility function to be maximized. There are two types of constraints: (i) *hard constraints*, which are required to be satisfied, and (ii) *soft constraints*, which should be satisfied 'as good as possible'. 'As good as possible' is defined as one or more variables to be taken into account in the objective function, or as a tolerance interval in which the solution has to be kept.

Constrained optimization is applied in many subdomains in robotics. For example, it is known as *constraint-based programming* [46, 137] in task specification and control; and it is known as *constraint satisfaction problems* [157] in artificial intelligence. Constrained optimization takes a central role in this dissertation as a mechanism to compose different functionalities.

This dissertation formulates all activities in robot applications as constrained optimization problems.

Activities that work together do not do that in traditional 'master-slave' or 'client-server' compositions (in which one activity sends 'setpoints' for the other one to realise), but in a 'peer-to-peer' mode in which both peers contribute to optimize a shared objective function, while at the same time keeping their (shared or non-shared) states away from constraints. The resources that contribute terms to, both, objective functions and constraints, are: the robot platforms with

their (finite) capabilities; the tasks with their (ideally infinite) requirements; the objects in the environment with their manipulation and perception affordances.

For example, ordering a task to be performed by a certain robot constrains both the task and the robot. The choice of the task limits (*constrains*) the behavior of the platform, and the choice of the platform limits –inter alia– the execution speed of the task. For example, robot butler Jeeves will be able to hand you the whiskey you requested. However, when requesting your autonomous car to bring you a whiskey, it will be able to drive to a liquor store, but it will not be able to hand you your drink. Both solutions are very different and impossible to provide parameterizations for, in a scenario that does not provide an enumeration of all possible task-robot combinations. The latter scenario is an *open world* scenario.

1.3.2 Hierarchy

Hierarchy provides a proven way to deal with complexity. It provides a single point of responsibility (the common parent in the hierarchy). However, it is a rigid structure, and causes problems in large and distributed systems if commands have to pass through different hierarchical levels.

The goal of this dissertation is to apply the concept of constrained-based peer-to-peer interaction also to the integration of hierarchical architectures: each level in the hierarchy provides objective functions and constraints to its higher and lower level peers for optimization, as well as requests *to be notified* (by events) if certain *tolerances are violated* on (possibly a larger set of) such objective functions and constraints.

Hence, *hierarchy is not used anymore to decouple behaviour in a strict way, and certainly not anymore to impose information hiding between hierarchical levels, but to structure the knowledge that is required at each level as the context in which functionality, coordination, composition, and scheduling are configured.*

For example, when Jeeves is out of whiskey, cross hierarchy communication and event broadcasting allows it to ask your autonomous car to drive it to a liquor store. Asking all your robots to clean up your house together forms another example. The common parent in the application hierarchy will manage conflicts in the cooperation between the robots.

1.3.3 Separation of concerns

Separation of Concerns [52] is a software design principle to modularize a system. The aim is to divide a system in individual parts, each of which addresses another –independent– concept (*concern*). One such approach is the *5Cs principle of separation of concerns* [32], which distinguishes five concerns: *communication, computation, coordination, configuration, and composition. The 5Cs principle of separation of concerns will be a key concept in this dissertation.*

For example, it is possible to detail and discuss the coordination of abovementioned cleaning task without the specification of how, when or where this task will get its configuration, how the task will be achieved (functionality, computations), what and how the robots will communicate, or what the robots are composed of.

1.3.4 Metamodeling

This dissertation applies metamodeling [20] to separate domain knowledge from its implementation in a software framework. It is an approach from Model Driven Engineering (MDE) that formalizes domain knowledge in a *meta-model*. This meta-model forms a modeling language, a *domain-specific language (DSL)*, that can be used to describe a specific *model*. In return, this model's *conformance* to the meta-model can be verified. An implementation in the software framework of choice can be hand-coded or generated from the model. When executed well, this approach should allow for easier code generation, verification, and debugging.

For example, it is possible to specify the cleaning example or the whiskey fetching example without specifying any software platform details. Whether the example is programmed using the ROS [130] or OROCOS [29] framework, will not limit us in modeling the application at hand. Moreover, it should be possible to generate (manually or automatically) an implementation for the application from its model.

1.3.5 Force-sensorless wrench control

In contrast to stiff industrial robots, service robots such as a PR2 robot (Figure 1.1) incorporate some compliance by design to ensure safe interaction with humans. Moreover, many service robot tasks do not require accurate force tracking control. Typical control schemes applied when the robot interacts with the environment include force and impedance control. However, the first requires

a force measurement, and the latter a torque- or acceleration-controlled robot, both not present on many service-robotic platforms. *The compliance and lower accuracy demands can be used, together with backdrivability, to develop a force-sensorless wrench control scheme that does not require accurate environment or robot models.*

For example, when James has to cut carrots, it has to apply a force on the carrots. However, the exact force applied on the carrots is of less importance. Moreover, it is an unrealistic requirement to provide a model for each object James has to apply a force on, in all possible environments.

Notwithstanding the general formulation of the research approach, this dissertation will not provide solutions for the cognitive or perception aspects of the outlined robot systems. Moreover, it does not intend to design new robot hardware, or provide multi-agent systems.

1.4 Contributions and outline

The major contributions of this dissertation help developers to deal with the increasing complexity of robotic applications. This complexity arises from the scale of the applications and their integration of different sub-systems and domain knowledge. It has triggered a need for knowledge-driven, flexible, reusable, and adaptable software, which is addressed in this dissertation. The key concept behind the major contributions can be summarized as the **Composition Pattern**. This pattern, based on model-based engineering, provides a concrete and constructive way to apply the 5Cs concept of separation of concerns. Figure 1.2 gives a schematic overview of the dissertation and its chapters.

The first contribution, presented in Chapter 3, consists of **the definition of the Composition Pattern and the systematic approach to apply the Composition Pattern to the modeling of robot applications.** The Composition Pattern provides the basic 'building block' to be made when designing application-specific, complex system architectures. However, it does not limit itself to only 'bringing functionality together', but adds following important application design concepts:[†]

- **Metamodeling** considers *'everything a model'*, in contrast to 'everything is an object' [20]. It gives the robot the possibility to reason, on- or off-line, about its own functionality, independent of the software framework or programming language of the implementation of the functionality.

[†]The structure, i.e. the actual pattern, and the related design concepts will be refered to –altogether– as the Composition Pattern.

- **Composition (Coordinator, Composer, Configurator, Scheduler, Communicator and Monitor, next to Functionality)**, forms a concrete structure and guideline to apply the 5Cs principle of separation of concerns. The chapter discusses how this improves the non-functional qualities such as reusability and flexibility.
- **Hierarchy and semantic context** guide the developer to divide a problem into sub-problems. The chapter examines how this improves the robustness, reusability, and adaptability of a design.

The chapter discusses how this methodology decreases the design pitfalls of rigidity, fragility, and immobility [103] and gives concrete guidelines on reuse and refactoring.

The second contribution, also presented in Chapter 3, consists of **the modelling of the domain of constraint-based programming using the Composition Pattern.** It describes a generic division of the domain of constraint-based programming: on the one hand, it describes the relation between meta-models; on the other hand, it describes the hierarchy of composite functional entities. The chapter focusses on detailing the task related aspects.

The third contribution, presented in Chapter 4, consists of **the development of a domain-specific language (DSL) for constraint-based programming**. This DSL provides a more user-friendly language with respect to manual code, guiding developers or users to create, adapt or understand an application. It forms a point of integration for existing and future modeling efforts, such as the Geometric Semantics DSL [42] and rFSM DSL for statecharts [90]. **Moreover, this chapter presents the software tools (i) to verify the conformity of concrete constraint-based application models to this DSL, and (ii) to transform the concrete models into an implementation.** This implementation is a run-time configuration and instantiation using the software framework discussed in Chapter 5.

The fourth contribution, presented in Chapter 5, consists of **the Composition Pattern as a *software architectural pattern*, and its application to the constraint-based programming software framework of iTaSC.** Moreover, this chapter discusses the lessons learned from refactoring the iTaSC software framework to a Composition Pattern compatible framework.

The fifth contribution, presented in Chapter 6, consists of **the development of a novel force-sensorless wrench control scheme for velocity controlled robots which does not require a precise dynamic model of the robot, environment, or contact point.** It is integrated in the resolved-velocity iTaSC approach and software framework, and allows the combination of the wrench control constraints with other constraints. Furthermore, it features a

reference adaptation factor, which can be applied to impose a desired transient behavior on the applied wrench. Experimental validation of the control scheme proves that a stable, constant contact wrench can be reached in a repeatable way, and with an accuracy that fits service robot tasks.

The sixth contribution, presented in Chapter 7, consists of **concrete applications developed using the abovementioned contributions, most particularly the complex use-case of force-sensorless and bimanual human-robot comanipulation.**

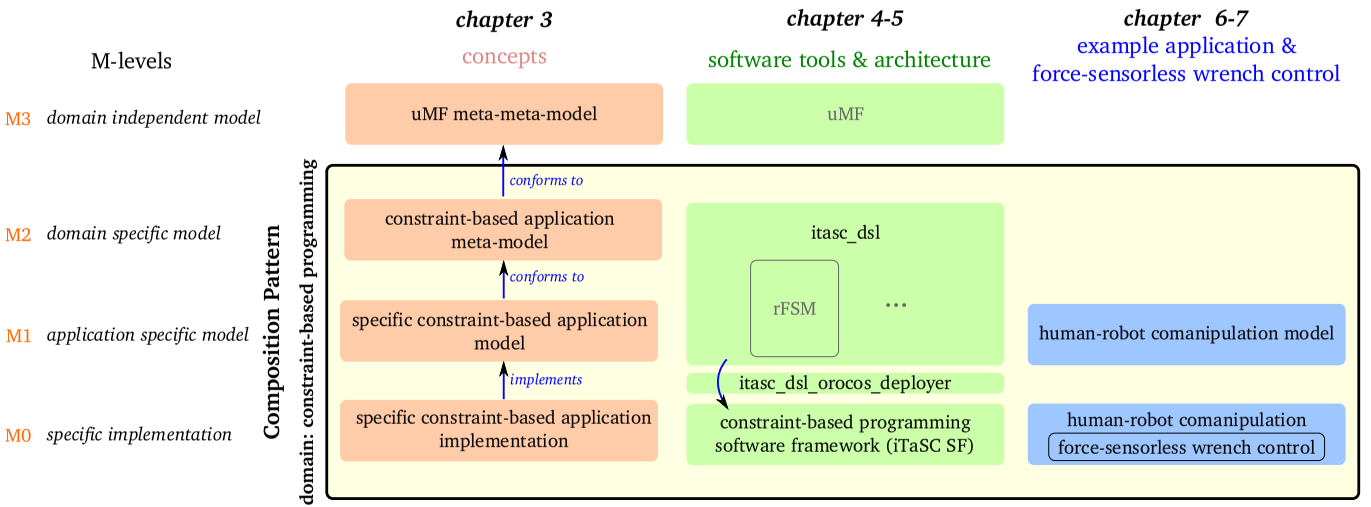


Figure 1.2: Overview of the dissertation. The parts in grey indicate existing DSLs or software, such as uMF and rFSM.

Chapter 2

Background and Positioning

Early robotic systems consisted of a single manipulator, where the end-effector had to complete a set of movements. They had limited functionality such as motion control or task sequencing, and hence structuring this behavior was only a modest problem. In the late 1960's, researchers at Stanford Research Institute created Shakey [118], a mobile robot that integrated a camera, a range finder, bump sensors, computers, and radio communication. It was, inter alia, able to plan a path from one location to another while avoiding obstacles. The study of the integration problem that arose, resulted in (one of) the first robot structural templates: sense-plan-act.

Throughout the evolution of robotics, more and diverse hardware was added to robot platforms. The robot platforms themselves also became more diverse in nature; evolving from a fixed robot arm to flying, driving, sailing, diving, or crawling platforms. Moreover, more types of functionality where developed, creating different sub-fields in robotics. This functionality includes controlling the motion of the robot, planning paths from one position to another, planning a sequence of tasks to execute, reasoning on the tasks to execute, coordinating tasks and behavior, and perceiving and modelling the outside world. With this explosion of behavior and hardware to integrate in a single system, the need for more thoughtful ways of structuring this integration emerged.

This chapter discusses different ways to structure and implement behavior. Furthermore, it discusses the motion control aspect of behavior, with a focus on Constraint-Based Optimization. Lastly, it discusses different ways of integrating Constraint-Based Optimization and related methodologies in a robot system. Seen the vast size of the field of robotics, this chapter does not intend to be exhaustive; it only discusses structure and behavior that influences or compares

to the contributions in this thesis. The Springer Handbook of Robotics [141] forms a good starting point to a more exhaustive overview.

2.1 Structuring and implementing behavior

Many approaches to structure, integrate and implement robot behavior exist. This section discusses some important, complementary categories of approaches.

2.1.1 Robot software architectures

Terminology

The definitions of the terminology describing how to structure behavior is not standardized; different, even conflicting definitions exists. This section gives an overview of the definitions used in this thesis. The definitions of Taylor et al. [152] serve as main reference for the applied terminology. The Software Engineering Institute of Carnegie Mellon University [150] gives an overview of known definitions.

Entity

This thesis considers an **entity** a concept or model that maps to software components, agents, objects, modules, processes, activities...

Design Decision

A **design decision** constitutes the choices made by a developer relating to system structure, functional behavior, interaction, non-functional properties, or implementation. Whether a design decision is a *principal* design decision depends on the goals and stakeholders [152].

Remark that design decisions are more than only 'structure'. Design decisions form an important concept for the definition of architecture.

(Software) Architecture

"A software system's **architecture** *is the set of principal design decisions made about the system. It is the blueprint for a software system's construction and evolution."* (Taylor et al. [152])

(Software) Reference Architecture

"A **reference architecture** *is the set of principal design decisions that are simultaneously applicable to multiple related systems, typically within an application domain, with explicitly defined points of variation."* (Taylor et al. [152])
In other words, a reference architecture forms a 'template' for a specific architecture in a certain application domain.

(Software) Architectural Pattern

"An **architectural pattern** *is a named collection of architectural design decisions that are applicable to a recurring design problem, parameterized to account for different software development contexts in which that problem appears. [...] It's a tactical design tool."* (Taylor et al. [152])

Related to architectural patterns are architectural styles.

(Software) Architectural Style

"An **architectural style** *is a named collection of architectural design decisions that are applicable in a given development context, constrain architectural design decisions that are specific to a particular system within that context, and elicit beneficial qualities in each resulting system. [...] It's a strategic design tool."* (Taylor et al. [152])

Architectural styles form a set of canonical architectural solutions to problems, they are underspecified architectures [14]. For example, the definitions classify the further on defined *client-server* as an architectural style, and *three-layered architectures* as an architectural pattern following the client-server architectural style.

The definitions above concern design aspects, and they are ordered from specific (architecture) to more general (architectural style). However, they do not specify specific code implementations. For implementations, the concept of a framework

is important.

(Software) (Architecture-implementation) Framework

"An architecture-implementation **framework** *is a piece of software that acts as a bridge between a particular architectural style and a set of implementation technologies. It provides key elements of the architectural style in code, in a way that assists developers in implementing systems that conforms to the prescriptions and constraints of the style. Frameworks may be extensive, requiring only minor portions to be completed by hand, or may be very basic, only offering help in implementing the most generic aspects of an architecture or architectural style. The places where hand work must be done are defined in advance. [...] A framework provides the bridge between concepts from the architecture and concepts from the platform (that is, programming language and operating system). [...] Frameworks encapsulate key features in mapping architectures to implementations."* (Taylor et al. [152])

The next paragraphs discuss different architectural patterns in the domain of robotics. Many architectures in robotics apply or combine aspects of these architectural patterns. The paragraphs give examples of (reference) architectures that apply the presented architectural patterns. A more in-depth discussion on different architectural patterns and reference architectures can be found in chapter eight of the Robotics Handbook [95], and in the book on software architectures by Taylor et al. [152].

Sense-Plan-Act

Sense-plan-act (SPA) [120] forms one of the earliest robot architectural patterns. It consists of three entities with a unidirectional flow between them: from *sense* over *plan* to *act*. The *sense* entity gathers sensor information and provides this to the *plan* entity. The *plan* entity uses the sensor information to update a world model, and generates thereafter a series of actions to execute. Subsequently, the *act* entity executes the planned actions.

This architecture proved to have issues concerning performance and scalability: firstly, due to the planning step between sensing and acting which blocks the robot, and secondly, due to the dangerous execution of a plan without tightly integrated sensing.

Subsumption

Brooks [28] defined the subsumption architectural pattern as a reaction to SPA. The subsumption architectural pattern abandons complete world models and plans. It consists of layers of interacting finite-state machines (FSM's), each connecting sensors to actuators directly. The entities, i.e. FSM's, of different layers influences each other's behavior through two operations: inhibition and suppression. The first prevents input to a FSM, the latter replaces the output of a FSM. This mechanism of inhibition and suppression 'selects' the active behavior and is referred to as *arbitration*.

The subsumption architectural pattern is highly modular and results in very reactive behavior. Each entity captures one part of the overall robot behavior, however there is no global representation (let alone, optimization) of the behavior: behavior emerges from the composition of the different entities. Therefore subsumption is also referred to as **behavioral or behavior-based robotics**. Subsumption provides layers, however does not provide explicit guidance or support on how to use these layers in the architecture [152]. The integration of long term goals proves to be difficult when using the subsumption architectural pattern.

Layered/Tiered Architectures

Layered architectures attempt to combine the advantages of the predictability of the plans of SPA, and the reactiveness of subsumption. It implies a separation of the software in layers of increasing abstraction, and typically increasing time scale. The most widely adopted layered architecture, 3T or 3L [64], defines three layers. The highest level, i.e. the *planning* or *decision* layer, performs long-term planning actions. The lowest level, i.e. the *reactive*, *functional* or *behavioral control* layer, performs motion control and fast (real-time) reactions to sensor information. The mid-level, i.e. the *sequencing* or *executive* layer, links the planning and reactive layer. It translates high-level goals to low-level actions. The information or data-flow between the layers is bi-directional.

As Taylor et al. point out [152], there is little architectural guidance to separate functionality into the three layers. Moreover, the sequencing layer in the middle of the architectural pattern can result in difficulties and performance overhead when translating the higher-level planning goals into low-level actions. Even more, many reference architectures tend to provide a layer that dominates the others in importance and complexity.

The **LAAS architecture** [4, 129], a three-layer reference architecture, introduces concrete behavior in 3T architectures, and provides software tools to

implement each layer.

The *functional layer* consists of a network of modules responsible for monitoring, control and perception. These modules are written using the GenoM (generator of modules) language, which produces standardized templates providing easy integration of modules in a GenoM-compliant network.

The *executive layer* forms a bridge between the functional and decision layer. It is purely reactive and does no task decomposition; it selects and parameterizes tasks to send to the functional layer. The executive layer is written in the Kheops language, to allow automatically generated decision networks that can be formally verified.

The *decision layer* consists of the IxTeT indexed time table temporal planner and scheduler [67] and the Procedural Reasoning Systems (PRS) supervisor [77]. The reference architecture allows for multiple decisional layers at increasingly higher levels of abstraction.

Another example of a layered reference architecture is **CLARAty** [115]. It is a two-layer reference architecture designed for the planetary rovers of NASA. The lowest, *functional layer* consists of a hierarchy of Object-Oriented modules providing interfaces of different levels of abstraction. The higher level objects provide platform independent functionality, the lower level object provide platform specific functionality. The highest, *decision layer*, merges the planning and executive layer of three-layer architectures. CLEaR [61] provides an instantiation of this decision layer, consisting of a planner/scheduler and a Task Description Language (TDL) [143]. This Task Description Language defines (high-level) tasks in a declarative way, which can be translated to C++ code.

Teleo-reactive architectures

The teleo-reactive architectural pattern defines different levels, each of which integrates planning and acting, in contrast to the layered architectures. The different levels operate at a different planning-acting horizon and time quantum [76]. Example reference architectures that conform to this architectural pattern, include T-ReX (Teleo-Reactive Executive) [105, 106] and IDEA [63, 68].

IDEA provides a unified planning and execution framework with a model-based multi-agent organization. It defines layers of agents where all agents share the same model representation primitives with the same semantics. An agent consists of a single model, plan database, and plan runner, and a variety of planners. The model provides a single locus of the known constraints and desired behaviors, usable by automatic reasoning systems. However the shortest

computing time quantum is in the order of a second, hence is not fast enough for 'low-level' control commands [76].

(Hierarchical) component-based, message-passing architectures

The architectural patterns mentioned above do not specify communication between the entities or layers of their patterns, in contrast to (hierarchical) component-based, message-passing architectures. **JAUS (Joint Architecture for Unmanned Systems)** [79] is an example hierarchical component-based, message-passing reference architecture. The composition of JAUS components, a node manager, and a communicator forms a node; the composition of nodes forms a subsystem; and the composition of subsystems forms a system. The node manager handles communication between nodes, and the communicator handles communication between subsystems. A subsystem represents a physical entity, e.g. a mobile robot. A node on the other hand, represents a computing device with a physical address. A component represents a 'service', eg. a visual sensor. Furthermore, JAUS provides a set of standardised interfaces and messages. For example the JAUS Service Interface Design Language (JSIDL) standardized by the Society of Automotive Engineers.

Different implementations and tools have been developed around the reference architecture. For example the JAUS Tool Set (JTS) [78], which provides open source software specification and development tools, as well as an open source software framework for JAUS.

2.1.2 Meta-modeling and separation of concerns

This thesis follows the *meta-modeling approach* of Model Driven Engineering (MDE), a systems design method going beyond the architecture-based method. The method aims at separating *domain knowledge* from implementation details. This domain knowledge is formalized in a modeling language, i.e. a *meta-model*. The modeling language can then be used to specify concrete *models*. In the other direction, models can be analyzed or validated using the meta-model. Moreover, a model can be transformed to executable code, or transformed to another model that uses another modeling language (model-to-model transformation).

There are two important modeling philosophies within MDE: the **Unified Modeling Language (UML)** [124] based **profiling** approach, and the *meta-modeling* approach [88]. UML *profiles* provide a mechanism to *extend and customize UML meta-models* for particular domains and platforms. The profile enables refining existing models in strictly additive manner, preventing them

from contradicting the existing model. The Object Management Group's (OMG) [122] robotics domain task force (Robotics-DTF) advocates profiling, and aims at robotic systems integration from modular entities through the adoption of OMG standards. This task force defined different standards, including the Robotics Technology Component (RTC) [123]. RTC forms the basis of a few framework ecosystems such as OpenRTM [7].

In contrast, the **meta-modeling** approach [20] encourages *creating new meta-models from scratch*. A meta-modeling language, i.e. a *meta-meta-model*, describes the concepts and relationships of the new meta-model. The Eclipse Modeling Framework (EMF) [57] follows this approach, providing the Ecore meta-modeling language. Klotzbücher introduced the micro-modeling language (uMF) [88, 92], the meta-modeling language which will be used in this thesis. In contrast to Ecore, uMF supports partially constrained, open world models.

uMF is an internal **domain-specific language (DSL)**. In contrast to a general purpose language, a domain-specific language is a language specifically designed to express the concepts of a particular domain. A DSL can be internal or external: an internal or embedded DSL is constructed on a host language, in constrast to an external DSL, which is developed from scratch. Lua [74] forms the host language for uMF.

The reduced Finite State Machine (rFSM) DSL [91] forms an example of a DSL conforming to uMF. It is a model of robotic tasks and systems coordination with a minimal set of semantic primitives. However, the use of a DSL does not imply using an MDE approach, DSLs are used in software engineering and robotics for a long time. Van Deursen et al. [160] give an overview of research on DSLs, and Klotzbücher [88] gives a (non-exhaustive) list of DSLs used in robotics.

An in depth discussion and overview of examples of the different MDE approaches can be found in the PhD thesis of Klotzbücher [88]. The major paradigm shift is that meta-modeling considers *all entities are formal models*, in contrast to the code-centric view of *all entities are code objects*.

Following paragraphs detail the terminology of the meta-modeling approach, inspired by the work of Bézivin [20].

Terminology

Model

A **model** captures a view or aspect of a system, it groups semantics. A model *conforms to* one or more meta-models.

The *'conforms to'* relation can be seen as synonymous to *'is a (domain-specific) language for'*. This thesis will indicate model names with the *italic* font. For example a *PR2* model describes the aspects of a system that are relevant for the domain of robotics where the model is intended to be used. It includes for example its kinematic and dynamic model, its available sensors, etc. Another model example includes a *pick and place task* model, describing a set of actions to pick an object and place it in a certain location.

Meta-Model

A **meta-model** presents a language to describe a model. A model *conforms to* a meta-model, a meta-model *conforms to* a meta-meta-model.

This thesis will indicate meta-model names with the `teletype` font. For example a `robot` meta-model provides a language to describe robots. It includes or composes a language to describe kinematic and dynamic models, available sensors, etc.

Bézivin [20] provides the analogy of a map to explain the differences between the meta-modeling concepts. A *map of Belgium* provides a model of the country of Belgium (the 'implementation'), the `legend of the map` provides the domain-specific language (DSL) in which the map is drawn. The legend states for example that a road is indicated by a white line, while a river is indicated by a blue line. This example shows also the difference of the very important *conforms to* relation with the *inherits from* relation between an object and a class in Object-Oriented programming.

Meta-Meta-Model

A **meta-meta-model** presents a language to describe a meta-model. A meta-model *conforms to* a meta-meta-model, a meta-meta-model possibly *conforms to* another meta-meta-model.

The abstract representation of a model needs to be transformed to executable code, i.e. the implementation.

Implementation

An **implementation** is a piece of software, it is an *instance of*, or *represented by* a model. From a model a concrete implementation can be generated or hand-coded.

M-levels

One of the most applied MDE approaches is the *Model Driven Architecture* (MDA) [109] by the Object Management Group. The Object Management Group [122] orders the presented terminology in four levels of model abstraction. Bruyninckx et al. [32] extend this concept to the domain of robotics, defining:

M0: The implementation of a robot application using a particular programming language.

M1: (Robot) platform specific models, i.e. the concrete models involved in a robot application.

M2: The meta-models, i.e. (robot) platform independent models divided in two sub-levels consisting of software framework specific meta-models and abstract 'Component-Port-Connector' meta-models.

M3: The meta-meta-model, where they applied the ECore meta-meta-model of the Eclipse Modeling Framework [57].

Klotzbücher et al. [93] use these levels of model abstraction to implement a domain-specific language for task specification using the Task Frame Formalism.* It is the platform dependency that constitutes the difference between the M1 and M2 levels of the BRICS meta-model.

This thesis uses the same four levels of model abstraction, however generalizes and extends the difference of the M1 and M2 levels. In the view of this thesis, a specific platform is represented at the M1 level by a model, next to other models of a robotic application such as a task† to achieve. Let us revisit the example of the the *PR2* robot model and the *pick and place task* model. Both models are independent of each other, but when brought together, they will

*Section 2.2 will discuss the Task Frame Formalism.
†Chapter 3 will define task in a more rigorous way.

have an impact on each other. In the philosophy and terminology of this thesis, we say they *constrain* each other: a robot's behavior is limited to execute the specific task, the task is limited in e.g. the execution speed by the specific robot. The *solving of the constraints* between the models to concrete parameters is considered a *model-to-model transformation* on the M1 level.

More concretely, this thesis denominates the M1 level as the *application-specific model*, M2 as the *domain-specific model*, and the M3 level as the *domain-independent model*, as shown in Figure 2.1.

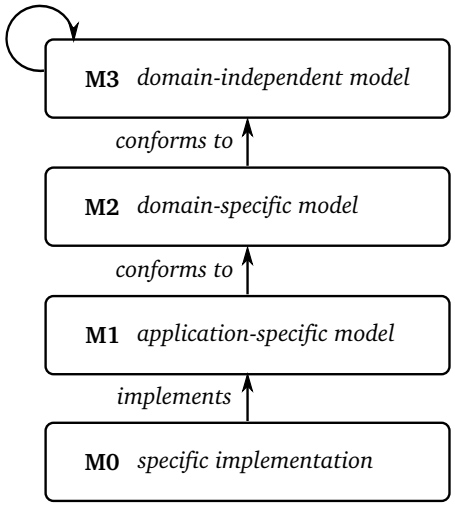


Figure 2.1: Definition of the four levels of model abstraction as defined in this thesis, and inspired by the MDE approach.

Separation of concerns

An important software design principle that aids a developer in designing software is *Separation of Concerns* [52]. The separation of concerns principle aims at dividing a system in individual parts that address independent concepts, i.e. *concerns*. It is a method to modularize a system, but it is in itself not *constructive*, i.e., it does not explain *how* to achieve the desired separation. This thesis follows the **5Cs principle of separation of concerns** [32] which considers the separation of the *communication, computation, coordination, configuration, and composition* functionality. It is an extension of the four concerns advocated by Radestock and Eisenbach [131] with the concern of

composition. This composition forms a key concern in this thesis. Bruyninckx et al. [32] define these concerns as follows:

- **Computation** is the core functionality of a system, which implements the domain knowledge.
- **Communication** is the functionality that brings data to the computational components.
- **Coordination** is the functionality that determines how the different components of a system work together and when they change their behavior.
- **Configuration** is the functionality to influence other components their behavior by changing parameter values.
- **Composition** is the functionality that determines how the other components (the other four C's) are coupled and how they interact.

The separation of computation, coordination and communication is common in many software frameworks. Klotzbüecher et al. [89] introduce *pure* coordination by separating configuration and coordination using the Coordination-Configuration Pattern. The work of Bruyninckx et al. [32] discusses the advantages of using the 5Cs approach, however does not deliver a concrete approach to achieve this separation in a systematic way. This thesis goes further by introducing the Composition Pattern as a constructive way to apply the 5Cs principle of seperation of concerns.

2.1.3 Software framework ecosystems

Software framework ecosystem

Software framework ecosystems provide a software development environment and the tooling to implement a certain architecture or application. The *ecosystem* addition refers to the fact that these frameworks form the centre of more specialized framework development or integration efforts.

An important set of software framework ecosystems use data flow or component-based techniques. This section regards two types of this set: modeling framework ecosystems, e.g. LabVIEW [117] or Simulink [154], and code-oriented framework ecosystems, e.g. ROS [130, 171], Orocos [33, 34], Orca [25, 26], JAUS Tool

Set [78], or RT-Middleware [6] implementations. RT-Middleware is a standard (reference architecture) for component-based frameworks using components that conform to the Robotics Technology Component (RTC) standard discussed in Section 2.1.2. An example instantiation of RT-Middleware is the OpenRTM-(AIST) framework, which itself is based on CORBA [125].

Michal et al. [108] and Elkady et al. [58] compare different implementation and tooling aspects of code-oriented framework ecosystems used in robotics. Philips [127] compares different code-oriented framework ecosystems based on the 5Cs principle of separation of concerns, and compares the constraints these frameworks pose on the software architecture. Chapter 3 will discuss different aspects of the framework ecosystems using the terminology of the 5Cs principle of separation of concerns.

2.1.4 Integration of approaches

The different approaches discussed in previous sections are not independent options, but can be used in a synergetic way. Section 2.1.1 indicated already the use of DSLs to describe architectures following an architectural pattern.

Bensalem et al. [19] provide another example based on BIP (Behavior, Interaction, Priority) [15]. BIP is a language for modeling real-time components, and it provides a framework to compose BIP components. BIP makes distinction between a system described in the BIP language and a BIP model. A system described in the BIP language can be parsed to a BIP model that conforms to the BIP meta-model. This model can be formally verified to detect possible deadlocks. Moreover, a code-generator generates C++ code from BIP models, which can be executed on a BIP Engine.

The BIP component model consists of atomic components that can be composed to compound components. An atomic component consists of ports, local variables, and a finite-state machine. Connectors specify interactions between the atomic components. Furthermore, priority rules describe scheduling policies for these interactions.

Bensalem et al. [19] integrated BIP with the LAAS architecture discussed in Section 2.1.1. It enables translating GenoM module specifications to BIP components (models). These BIP models can then be formally verified, and code can be generated. This approach is further extended to use the connectors to define constraints [18]. Abdellatif et al. [2] introduces a DSL to specify these constraints, which can be transformed to the connector specification.

This is only one example of many concrete architectures and approaches in

robotics, but it comes close to the methodology advocated in this thesis. One important set of approaches, which has recently seen a revival of interest, are Artificial Intelligence (AI), and in particular **knowledge driven, approaches**. Ingrand and Ghallab [76] give an overview of these approaches, dividing them based on the deliberation functions of perceiving (sensing), planning, acting, monitoring, and goal reasoning. The authors further distinguish between different acting approaches: procedure based, automata based, logic based, constraint satisfaction problem based, and stochastic based approaches.

The category of **automata based approaches** is larger than only artificial intelligence methods, relating to the coordination and configuration of motion. Examples include Skill/Manipulation Primitive Nets [62,155], which coordinate and configure hybrid force/position controllers and their setpoints. LightRocks [156] extends this work by introducing levels of abstraction of task specifications and a supporting DSL. The latter relates to the Task-Skill-Motion approach of Smits et al. [144,145,147] applied to constraint-based programming, which will be detailed in Sections 2.2 and 2.3. In the category of automata based approaches, the work of Klotzbücher et al. [88,91] on coordination and configuration of robot systems will be influential to this thesis, as will be detailed in Chapters 3 to 5.

In the set of knowledge-driven approaches, Doherty et al. [54] provide an example of coherent integration of functionality across the different deliberation functions. Their approach enables automatic generation, specification, verification, and execution of high-level Unmanned Aerial Vehicle (UAV) missions. It integrates a temporal logic based mission specification language, a distributed temporal planner, a task specification language, and an agent-based software architecture within the ROS framework ecosystem.

This thesis aims at supporting the design and integration challenge discussed in this section. Moreover, it does so in a uniform way, across the different deliberation functions by introducing the Composition Pattern, which will be elaborated in Chapter 3.

2.2 Motion and force control using Constraint-Based Optimization[‡]

An important part of behavior in robotics results from motion and force control. This section first describes the evolution of motion and force control in robotics,

[‡]This section is partially based on Vanthienen, D., De Laet, T., Decré, W., Bruyninckx, H., De Schutter, J. (2012), "Force-Sensorless and Bimanual Human-Robot Comanipulation", *10th IFAC Symposium on Robot Control*, Dubrovnik, Croatia, 5-7 September 2012 (pp. 1-8)

resulting in constraint-based optimization. The section then focusses on the iTaSC approach to constraint-based optimization, developed in our research group. Lastly, the section discusses approaches to solving the optimization problem.

2.2.1 Motion and force control

The motion of a robot consists of free-space movements and movements in contact with the environment, i.e. compliant motion.

Compliant motion

There are two main approaches to compliant motion: *hybrid (force-position) control*, and *impedance control*. Hybrid force-position control [132] decomposes the robot workspace in n force controlled directions, and $6-n$ motion controlled directions. Impedance control (or admittance control) [70–72, 85] on the other hand, is the generalization of stiffness, compliance [136], and damping control [170]. Impedance control imposes a desired mechanical impedance behavior of the robot in contact with the environment. Where hybrid control regulates a force or position setpoint, impedance control regulates the relation between both, i.e. the impedance.

Other approaches search to combine both approaches in different ways. Parallel force-position control [37], or 'feedforward motion in a force controlled direction' [44, 48], relates to hybrid control, however allows the weighted or prioritized combination of force and motion control in a single direction. This weighting or prioritization leads to a force and/or position error, as will be discussed in Section 2.2.3. Anderson and Spong introduce hybrid impedance control [5], which controls force and position while monitoring impedance. It is another way of bringing hybrid and impedance control together.

Task Space Formalism and Operational Space Formalism

The Compliance Frame [104, 132] and **Task Frame Formalism** (TFF) [31, 47] provide task specification support for hybrid force-position control expressed in a *compliance* or *task frame*. In this frame, different control modes, such as trajectory following, force control, visual servoing [11] or distance control, are assigned to each of the translational directions along the frame axes and to each of the rotational directions about the frame axes. For each of these control modes a setpoint or a desired trajectory is specified.

Khatib [86] extends the compliance frame formalism approach to the **Operational Space** motion specification, which includes the dynamic model of the robot, and also allows for the specification of joint-space motions.

For (geometrically) simple tasks, and 6-DOF robots, the Task Frame based approach has proven its effectiveness. However, this approach scales poorly to more complex tasks and robots resulting in a variety of ad-hoc implementations. Examples of these more complex tasks, not fitting in the task frame approach, are control of multi-point contacts [31] and bimanual robot manipulation, which is the topic of Chapter 7.

Constraint-based programming

Constraint-based programming takes a conceptually different approach. It does not consider the robot joints nor the task frame *central* to the formulation. Instead, the core idea behind the approach is to describe a robot task as a set of constraints, which do not necessarily have to be formulated in a single task frame, and a set of objective functions. Two such approaches are presented by Samson et al. [137] and De Schutter et al. [46]. The latter approach, denoted **i**nstantaneous **Ta**sk **S**pecification using **C**onstraints (**iTaSC**), introduces a set of 'auxiliary'[§] coordinates to express task constraints and model geometric uncertainty. Decré et al. [51] extended iTaSC first to support inequality constraints, then to a general optimization problem formulation [49], and later to time-independent trajectories and user-configurable task horizons [50]. Borghesan et al. [23] formalizes hybrid position-impedance-force tasks using the iTaSC approach.

The key advantages of constraint-based programming over classical motion specification methodologies are: (i) *composability of constraints*: multiple constraints can be combined and they can be partial, hence they do not have to constrain the full set of degrees-of-freedom (DOF) of the robot system; (ii) *reusability of constraints*: constraints specify a relation between frames attached to objects that have a semantic meaning in the context of a task, therefore the same task specification can be reused on different objects. iTaSC, in addition, allows the (i) *derivation of the control solution*: the iTaSC workflow provides a systematic approach to obtain expressions for the task constraints, to evaluate these expressions at run time, and to generate a robot motion that complies with the constraints by automatically deriving the input for a low-level controller; (ii) *modelling of uncertainty*: it provides a systematic approach to model uncertainties in this geometric model, to estimate these uncertainties at run time, and to adapt the control solution accordingly.

[§]Auxiliary with respect to the robot coordinates.

Constraint-based programming, and the special cases of Task Frame Formalism and Operational Space approach, all formulate task specification as a *constrained optimization problem*. They specify sets of constraints in a generalized *task space*, whether defined as a configuration, sensor, manipulation, constraint, task, or feature space.[¶] Constraints can be realised using different types of control schemes, and can be solved at velocity, acceleration, or torque level. The composition of these constraints and a set of objective functions, forms a numerical optimization problem. A *solver algorithm* solves this optimization problem, instantaneously or over a time horizon, and produces instantaneous robot motion setpoints, e.g. joint velocities or accelerations.

Constraint-based programming can specify motion and force constraints. A broad overview of force control schemes can be found in different review papers [35, 45, 113, 176]. Chapter 6 will discuss certain aspects of force control schemes in detail.

2.2.2 instantaneous Task Specification using Constraints (iTaSC)

Although the main contributions of this thesis are generally applicable to software frameworks for robotics, the iTaSC approach will take a central role in this thesis. It takes this position since the contributions evolved from the experiences in the development and usage of a software framework for the iTaSC approach.

The *iTaSC approach* to task specification and control [46] advocates a systematic way to construct a task space, i.e. the *iTaSC workflow*. The *iTaSC software framework* gives software and tool support to this workflow, exploiting the features of the Orocos component framework ecosystem, as explained in Section 2.3.

iTaSC workflow*

An iTaSC application consists of tasks, robots and objects, a scene-graph, and a solver.

For every application, the programmer first has to identify the **robots and objects**. Next, the kinematic structure of the robots and objects have to be

[¶]This thesis will use *task space* to denote the generalized task space. It will use a specific term when a more restricted context is needed.

*To prevent overloading the notation, this section leaves out uncertainty coordinates; perception and uncertainty resolution are not in the focus of this thesis, but can be seemlessly integrated in the presented workflow.

defined. They start at a reference frame (called base frame $\{b\}$) of the robot or object. The state of the kinematic structure is determined by the *joint coordinates* $\boldsymbol{q}$. Next, the programmer defines *object frames* $\{o\}$ on the robots and objects (i.e. frames on their kinematic structure) at locations where a task will take effect, for instance the robot end effector or an object to be tracked.

The actual **tasks** define the space between pairs of object frames ($\{o1\}$ and $\{o2\}$), i.e. the *feature space* or the generalized task space, represented by a *task space representation (TSR)*. An explicit formulation of this TSR can be seen as a *virtual kinematic chain (VKC)* between the object frames. To simplify the task definition, *feature frames* are introduced [46]. The feature frames are linked to a *physical entity* on a robot or object (such as a vertex or surface), or an *abstract geometric property* of a physical entity (such as the symmetry axis of a cylinder). Each task needs four frames: *two object frames* (called $\{o1\}$ and $\{o2\}$, each attached to a robot or object), and *two feature frames* (called $\{f1\}$ and $\{f2\}$, each attached to one of the corresponding features on an object or robot). For an application in 3D space, there are in general six DOF between $\{o1\}$ and $\{o2\}$. By introducing the feature frames, they are distributed over three subspaces shown in Figure 2.2: (i) subspace I, between $\{f1\}$ and $\{o1\}$ (feature coordinates $\boldsymbol{\chi_{fI}}$), (ii) subspace II, between $\{f2\}$ and $\{f1\}$ (feature coordinates $\boldsymbol{\chi_{fII}}$), and (iii) subspace III, between $\{o2\}$ and $\{f2\}$ (feature coordinates $\boldsymbol{\chi_{fIII}}$).

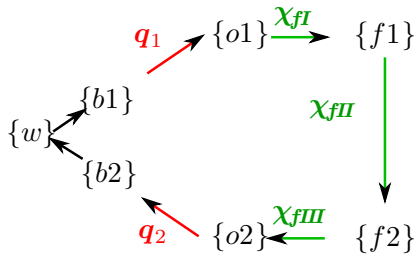


Figure 2.2: General kinematic loop related to a task. The red lines represent the kinematic structure of the robot and objects, the black line represents the (fixed) relation between the robot's fixed reference frame ($\{b\}$) and the world reference frame ($\{w\}$) defined in the scene-graph, and the green lines represent the *TSR* (e.g. *VKC*) between the object frames ($\{o1\}$ and $\{o2\}$).

To obtain the desired task behavior (motion), one has to **impose constraints** on the relative motion between the two objects. To this end, the programmer has to choose the outputs that have to be constrained by defining an *output equation*: $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{q}, \boldsymbol{\chi_f})$. De Schutter et al. [46] provide guidelines on how to define a VKC for a task such that the outputs are simple functions, in most cases simple selectors, of the feature and joint coordinates. The **imposed constraints** used

to specify the task are then directly expressed on the outputs as: $\boldsymbol{y} = \boldsymbol{y}_d$, for equality constraints, or $y \geqslant y_d$ or $y \leqslant y_d$, for inequality constraints. Each output constraint is enforced by a **controller**, which receives the desired output values ($\boldsymbol{y}_d$) from a *set-point generator*.

By defining the relations between the reference frames of the robots and objects and a global world reference frame $\{w\}$ in the **scene-graph**, the programmer defines how the robots and objects are located in the application scene. By connecting the VKC of the tasks to the object frames on the robots and objects, the programmer defines which robots execute the tasks on which objects. As such, each task defines a kinematic loop in the scene as shown in Figure 2.2. The kinematic loops introduce constraints between the robot coordinates $\boldsymbol{q}$ and the feature coordinates $\boldsymbol{\chi_f} = \left[\boldsymbol{\chi_{fI}}^T, \boldsymbol{\chi_{fII}}^T, \boldsymbol{\chi_{fIII}}^T\right]^T$, expressed by the loop closure equation: $\boldsymbol{l}\left(\boldsymbol{q}, \boldsymbol{\chi_f}\right) = \boldsymbol{0}$.

The **solver** provides a solution for the optimization problem of calculating the desired robot joint values (i.e. the joint velocities $\dot{\boldsymbol{q}}_d$ for a velocity-based control scheme) out of the task constraints. This allows to take into account different task priorities, different task constraint weights (in the case of conflicting constraints, i.e. overconstrained), and weights for the joints of the robots (to solve the kinematic redundancy of the robot in the underconstrained case).

2.2.3 Solving the constrained optimization problem

Constrained optimization is a well studied domain in mathematics. An optimization problem tries to find a solution, in this case an input to the robot, that satisfies all *constraints*, and that minimizes an *objective function*. The objective function is a minimization criterion. Example minimization criterion include the minimization of the input to the robot, and the minimization of the error on a soft constraint, i.e. a constraint that can be deviated from such as a freespace trajectory. The minimization of the control input to the robot can be the minimum norm in case the units of its DOF are the same. However a better solution, especially when the output DOF have different units, is a weighted norm with physical meaning, such as the minimization of kinetic energy in the resolved-velocity case, by using the inertia or mass matrix as weight [55].

An example of a general formulation of constrained optimization in robotics can be found in the work of Decré et al. [50, 51]. This formulation can define a global optimum, i.e. optimal over a time horizon or over a whole trajectory, or a local optimum, i.e. optimal only at a single time instance, considering the current task setpoints. However, one of the most applied optimization problem formulations for the resolved-velocity case in real-time robotics is the set of linear

equations posed by the differential kinematics equation $\dot{\boldsymbol{y}} = \boldsymbol{J_q}\dot{\boldsymbol{q}}$. This equation is characterized by the Jacobian matrix $\boldsymbol{J_q}$, or more generally, the augmented Jacobian $\boldsymbol{A}$. The latter is defined in the iTaSC context as the matrix that relates the desired joint velocity $\dot{\boldsymbol{q}}_d$ to the desired task space velocity after control $\dot{\boldsymbol{y}}_d^\circ$, for all tasks combined. This optimization problem is commonly solved locally and in the least-squares sense using the **Moore-Penrose pseudo-inverse** of the augmented Jacobian [126], i.e. $\dot{\boldsymbol{q}}_d = \boldsymbol{A}^{\#}\dot{\boldsymbol{y}}_d^\circ$. This pseudo-inverse method minimizes the norm of the output $\| \dot{\boldsymbol{q}}_d \|$ and the norm of the error on the constraints $\| \boldsymbol{A}\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{y}}_d^\circ \|$.

Different situations of the optimization problem are possible, which we will examplify using the differential kinematics equation problem formulation. Considering the augmented Jacobian matrix $\boldsymbol{A}$ of size $m \times n$, and Rouché's theorem, following situations [134] can be discerned:

- $rank(\boldsymbol{A}) = rank(\boldsymbol{A} \;\; \dot{\boldsymbol{y}}_d^\circ) = n$: The robot is *exactly constrained*, there is a unique solution $\dot{\boldsymbol{q}}_d$. For example a non-redundant, six axis robot that has to complete a task with its end-effector in a six DOF task space is an exactly constrained problem.
- $rank(\boldsymbol{A}) = rank(\boldsymbol{A} \;\; \dot{\boldsymbol{y}}_d^\circ) < n$: The robot is *underconstrained*, there is an infinite number of solutions $\dot{\boldsymbol{q}}_d$. For example a redundant robot that has to complete a task with its end-effector in a six DOF task space is an underconstrained problem.
- $rank(\boldsymbol{A}) = n$ and $rank(\boldsymbol{A}) \neq rank(\boldsymbol{A} \;\; \dot{\boldsymbol{y}}_d^\circ)$: The robot is *overconstrained*, there is no solution $\dot{\boldsymbol{q}}_d$ that satisfies all constraints. The objective function of the optimization problem 'arbitrates' the conflicting constraints. For example a non-redundant, six axis robot that has to complete a task with its end-effector in a six DOF task space while constrained to hold a certain joint configuration, is an overconstrained problem.
- $rank(\boldsymbol{A}) < n$ and $rank(\boldsymbol{A}) \neq rank(\boldsymbol{A} \;\; \dot{\boldsymbol{y}}_d^\circ) < n$: The robot is *under- and overconstrained*, there is no solution $\dot{\boldsymbol{q}}_d$ that satisfies all constraints, however there is an infinite number of solutions that minimize the error $\| \boldsymbol{A}\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{y}}_d^\circ \|$. For example a robot that has to complete two tasks with its end-effector, each constraining the same Cartesian direction, is an under- and overconstrained problem.

Dealing with conflicting constraints

The pseudo-inverse method treats conflicting constraints equally, hence the solution will be in the middle of the conflicting constraints. There are two

common methods to influence the arbitration of these conflicting constraints, task **weighting** [55] and **prioritization** [17, 69, 100, 112]. The first uses a weighted norm of the constraint error $\| \boldsymbol{A}\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{y}}_d^\circ \|_{\boldsymbol{W}_y}$, similar to the weighted norm of the solution vector $\| \dot{\boldsymbol{q}}_d \|_{\boldsymbol{W}_q}$ discussed above. Remark that the task weights $\boldsymbol{W}_y$ have no effect if $\boldsymbol{A}$ is exactly constrained.

Task prioritization on the other hand, projects secondary tasks in the null space of primary tasks, and tries to satisfy them in this reduced space, to the extent that the highest priority task will allow without being disturbed by it. This approach can be extended to multiple priority levels, e.g. tertiary tasks can be projected in the null space of the combined primary and secondary tasks etc. An overview of different formulations of both methods can be found in [9]. The two methods can be combined, resolving some conflicts using weighting, while resolving others using prioritization; this is, for example, done in the iTaSC realization by Rutgeerts [134].

Dealing with singularities

An important problem that needs to be avoided are **singular configurations** of the kinematic structures used in the task specification, in the first place the robot. Singular configurations occur when the augmented Jacobian 'looses rank', i.e. when the $rank(\boldsymbol{A}) < min(m, n)$. For example when two axes of a six DOF robot align, the robot will be unable to move instantaneously in one of the directions of the six DOF Cartesian space. The Moore-Penrose pseudo-inverse will result in a robot command of an infinite magnitude to try to move in this instantaneously unattainable direction.

Singularities can be avoided by introducing extra constraints, or using an appropriate objective function that avoids large robot commands in the neighborhood of singularities. A possible approach is to apply the pseudo-inverse method, but modify it when 'close' to singularities. Moreover, this modification should result in feasible and continuous robot commands. To quantify this 'closeness' to a singularity, a measure of distance to the singularity is required. Typical configuration dependent measures of distance to singularity are manipulability, the condition number, or the smallest singular value [36].

An example modified pseudo-inverse method is **regularization** or **damped least-squares** [111, 169]. It consideres a trade-off between tracking accuracy and the norm of the resulting robot commands, characterized by a damping factor λ:

$$\min_{\dot{\boldsymbol{q}}_d}(\| \dot{\boldsymbol{y}}_d^\circ - \boldsymbol{A}\dot{\boldsymbol{q}}_d \|^2 + \lambda^2 \| \dot{\boldsymbol{q}}_d \|^2). \tag{2.1}$$

The choice of the damping factor is important: a small value has low robustness near singularities, a large value has low tracking accuracy. Different methods to select the damping factor are proposed in literature [87, 111, 169].

One of the methods to choose a damping factor [101] considers (i) the smallest non-null singular value along a given trajectory or at a time instant as a measure of distance to a singularity, and (ii) the maximum allowed motor command, hence the minimum damping needed to ensure feasible motor commands. The approach further applies adaptive damping; it increases the damping when close to singularity, while it disables damping when far from singularities in order to avoid performance degradation.

Prioritized, weighted, damped pseudo-inverse

Baerlocher [9, 10] combined task prioritization and regularization in a computationally efficient, recursive algorithm. Rutgeerts [134] on the other hand, combined task prioritization and weighting. This thesis combines the approach of Baerlocher and Rutgeerts, i.e. it combines prioritization, task weighting, and regularization, extending the efficient algorithm of Baerlocher with weights.

Consider the division of the tasks into (i) primary constraints, characterized by the Jacobian $\boldsymbol{J}_{\boldsymbol{q}1}^{\#}$ and desired output $\dot{\boldsymbol{y}}_{d1}^{\circ}$, and (ii) secondary constraints, characterized by the Jacobian $\boldsymbol{J}_{\boldsymbol{q}2}^{\#}$ and desired output $\dot{\boldsymbol{y}}_{d2}^{\circ}$. Expressing this two-level prioritization using the formulation by Nakamura, Hanafusa and Maciejewski results in [69, 100, 112]

$$\dot{q} = \boldsymbol{J}_{\boldsymbol{q}1}^{\#}\dot{\boldsymbol{y}}_{d1}^{\circ} + [\boldsymbol{J}_{\boldsymbol{q}2}(\boldsymbol{I} - \boldsymbol{J}_{\boldsymbol{q}1}^{\#}\boldsymbol{J}_{\boldsymbol{q}1})]^{\#}(\dot{\boldsymbol{y}}_{d2}^{\circ} - \boldsymbol{J}_{\boldsymbol{q}2}\boldsymbol{J}_{\boldsymbol{q}1}^{\#}\dot{\boldsymbol{y}}_{d1}^{\circ}). \tag{2.2}$$

It is the simplification of

$$\dot{q} = \boldsymbol{J}_{\boldsymbol{q}1}^{\#}\dot{\boldsymbol{y}}_{d1}^{\circ} + (\boldsymbol{I} - \boldsymbol{J}_{\boldsymbol{q}1}^{\#}\boldsymbol{J}_{\boldsymbol{q}1})[\boldsymbol{J}_{\boldsymbol{q}2}(\boldsymbol{I} - \boldsymbol{J}_{\boldsymbol{q}1}^{\#}\boldsymbol{J}_{\boldsymbol{q}1})]^{\#}(\dot{\boldsymbol{y}}_{d2}^{\circ} - \boldsymbol{J}_{\boldsymbol{q}2}\boldsymbol{J}_{\boldsymbol{q}1}^{\#}\dot{\boldsymbol{y}}_{d1}^{\circ}) \tag{2.3}$$

$$= \dot{\boldsymbol{q}}_{d1} + \boldsymbol{P}_{N(1)}\dot{\boldsymbol{q}}_{d2} \tag{2.4}$$

since the null space projector $\boldsymbol{P}_{N(1)} = \boldsymbol{I} - \boldsymbol{J}_{\boldsymbol{q}1}^{\#}\boldsymbol{J}_{\boldsymbol{q}1}$ is idempotent and Hermitian. However, in case the pseudo-inverse is replaced by a weighted pseudo-inverse, with weights $\boldsymbol{W}_q$ on the controllable DOF, this condition does not hold and the null space projector should be reincluded. Introducing task weights $\boldsymbol{W}_y$

and robot joint weights $\boldsymbol{W}_q$, equation (2.3) becomes

$$\begin{aligned}\dot{q} &= \boldsymbol{J}^{\#}_{\boldsymbol{q}1,\boldsymbol{W}_{1q,y}}\dot{\boldsymbol{y}}^{\circ}_{d1} \\ &+ (\boldsymbol{I} - \boldsymbol{J}^{\#}_{\boldsymbol{q}1,\boldsymbol{W}_{1q,y}}\boldsymbol{J}_{\boldsymbol{q}1})[\boldsymbol{J}_{\boldsymbol{q}2}(\boldsymbol{I} - \boldsymbol{J}^{\#}_{\boldsymbol{q}1,\boldsymbol{W}_{1q,y}}\boldsymbol{J}_{\boldsymbol{q}1})]^{\#}_{\boldsymbol{W}_{2q,y}}(\dot{\boldsymbol{y}}^{\circ}_{d2} - \boldsymbol{J}_{\boldsymbol{q}2}\boldsymbol{J}^{\#}_{\boldsymbol{q}1,\boldsymbol{W}1_{q,y}}\dot{\boldsymbol{y}}^{\circ}_{d1}),\end{aligned} \tag{2.5}$$

as formulated by Rutgeerts [134]. Including regularization with damping factor λ, equation (2.5) becomes

$$\begin{aligned}\dot{q} &= \boldsymbol{J}^{\#}_{\boldsymbol{q}1,\lambda,\boldsymbol{W}_{1q,y}}\dot{\boldsymbol{y}}^{\circ}_{d1} \\ &+ (\boldsymbol{I} - \boldsymbol{J}^{\#}_{\boldsymbol{q}1,\boldsymbol{W}_{1q,y}}\boldsymbol{J}_{\boldsymbol{q}1})[\boldsymbol{J}_{\boldsymbol{q}2}(\boldsymbol{I} - \boldsymbol{J}^{\#}_{\boldsymbol{q}1,\boldsymbol{W}_{1q,y}}\boldsymbol{J}_{\boldsymbol{q}1})]^{\#}_{\lambda,\boldsymbol{W}_{2q,y}}(\dot{\boldsymbol{y}}^{\circ}_{d2} - \boldsymbol{J}_{\boldsymbol{q}2}\boldsymbol{J}^{\#}_{\boldsymbol{q}1,\lambda,\boldsymbol{W}1_{q,y}}\dot{\boldsymbol{y}}^{\circ}_{d1}).\end{aligned} \tag{2.6}$$

Remark that this equation avoids damping of the pseudo-inverse in the projection, as proposed and detailed by Baerlocher [9,10].

As discussed above, the introduction of the task weights $\boldsymbol{W}_y$ in the pseudo-inverse results in the minimization of the weighted norm of the task constraint error. This is trivial in the case of the primary constraint. For the secondary constraint, this results in

$$\left\|(\dot{\boldsymbol{y}}^{\circ}_{d2} - \boldsymbol{J}_{\boldsymbol{q}2}\boldsymbol{J}^{\#}_{\boldsymbol{q}1,\lambda,W_{1q,y}}\dot{\boldsymbol{y}}^{\circ}_{d1}) - (\boldsymbol{J}_{\boldsymbol{q}2}(\boldsymbol{I} - \boldsymbol{J}^{\#}_{\boldsymbol{q}1,W_{1q,y}}\boldsymbol{J}_{\boldsymbol{q}1})\dot{q}_2)\right\|_{\boldsymbol{W}_y}, \tag{2.7}$$

which can be rewritten as

$$\left\|\dot{\boldsymbol{y}}^{\circ}_{d2} - \boldsymbol{J}_{\boldsymbol{q}2}(\boldsymbol{J}^{\#}_{\boldsymbol{q}1,\lambda,W_{1q,y}}\dot{\boldsymbol{y}}^{\circ}_{d1} + (\boldsymbol{I} - \boldsymbol{J}^{\#}_{\boldsymbol{q}1,W_{1q,y}}\boldsymbol{J}_{\boldsymbol{q}1})\dot{q}_2)\right\|_{\boldsymbol{W}_y}, \tag{2.8}$$

or

$$\left\|\dot{\boldsymbol{y}}^{\circ}_{d2} - \boldsymbol{J}_{\boldsymbol{q}2}(\dot{q}_1 + \boldsymbol{P}_{N(1),\boldsymbol{W}_{1q,y}}\dot{q}_2)\right\|_{\boldsymbol{W}_y} = \left\|\dot{\boldsymbol{y}}^{\circ}_{d2} - \boldsymbol{J}_{\boldsymbol{q}2}\dot{q}_{,1..2}\right\|_{\boldsymbol{W}_y}. \tag{2.9}$$

In this equation $\boldsymbol{P}_{N(1),\boldsymbol{W}_{1q,y}}$ denotes a projection matrix in the null space of the Jacobian of the primary constraints using weights, and $\dot{q}_{,1..2}$ denotes the resulting joint velocity when taking primary and secondary constraints into account. The equation minimizes the weighted norm of the error on the secondary constraints, hence results in the desired behavior.

Baerlocher [9, 10] extends the formulation of damped least-squares to multiple priority levels using a computationally efficient, recursive formulation. This thesis complements this formulation with joint

velocity command weighting and task weighting, including the null space projection reintroduced in equation (2.3):

$$\dot{\boldsymbol{q}}_{d,1..i} = \dot{\boldsymbol{q}}_{d,1..i-1} + \boldsymbol{P}_{N(i-1)}(\boldsymbol{J}_{\boldsymbol{q}i}\boldsymbol{P}_{N(i-1)})^{\#}_{\lambda,W_{iq,y}}(\dot{\boldsymbol{y}}^{\circ}_{di} - \boldsymbol{J}_{\boldsymbol{q}i}\dot{\boldsymbol{q}}_{d,1..i-1}) \tag{2.10}$$

$$\boldsymbol{P}_{N(i)} = \boldsymbol{P}_{N(i-1)} - (\boldsymbol{J}_{\boldsymbol{q}i}\boldsymbol{P}_{N(i-1)})^{\#}_{\lambda,W_{iq,y}}\boldsymbol{J}_{\boldsymbol{q}i}\boldsymbol{P}_{N(i-1)}, \tag{2.11}$$

with

$$\boldsymbol{P}_{N(0)} = \boldsymbol{I} \tag{2.12}$$

$$\dot{\boldsymbol{q}}_{d,0} = \boldsymbol{0}. \tag{2.13}$$

$\boldsymbol{P}_{N(i)}$ denotes the projection in the null space of the ith Jacobian, $\dot{\boldsymbol{q}}_{d1..i}$ denotes the desired joint velocity as a result from all constraints from the first till the ith priority level.

The combination of weighting, regularization, and prioritization should be used with caution in the case that one of the priority levels is an under- and overconstrained problem where $rank(\boldsymbol{J}_{\boldsymbol{q}^i}) < m$ and $rank(\boldsymbol{J}_{\boldsymbol{q}i}) < n$. In this case $\boldsymbol{J}_{\boldsymbol{q}i}$ is not of full column nor row rank, it is a singular matrix. The damping will avoid the singular matrix condition, although the singularity can be caused by the conflicting constraints instead of a singular robot configuration. As a result the resulting $\dot{\boldsymbol{q}}_{d,1..i}$ will be inaccurate, on top of the constraint weighting. This situation is not present in the use cases and examples discussed in this thesis, since all joints have a secondary joint configuration constraint.

Chapter 6 and 7 will detail use cases of the described prioritized, weighted, damped pseudo-inverse solver. An implementation [161] in the C++ language has been developed and made public under an open-source license.

2.3 Software support for motion and force control

This section discusses the software tools that arose around the motion and force control approaches discussed in previous section.

Different software tools to support the Task Frame Formalism have been developed. The Compliant Motion Research and Development Environment (COMRADE) [159, 174, 175] provided one of the first DSLs for TFF. A COMRADE application consists of three entities that need to be detailed by the application developer:

- *the model*, i.e. the specification of the Task Frame and the selection of the force- and velocity-controlled directions,

- *the generator*, i.e. the selection of the desired setpoint for each force and velocity controller, and
- *the Discrete Event System*, i.e. the specification of the conditions that terminate a motion, and a befitting state change.

Skill/Manipulation Primitive Nets [62, 98, 155], already mentioned in Section 2.1.4, focusses on the coordination and configuration of hybrid force/position controllers and their setpoints. Recently, the Light-weight Robot Coding for Skills (LightRocks) approach [156] extends this work with different levels of abstraction of task specifications. LightRocks provides a DSL and code generation.

Klotzbücher et al. [93] introduce a meta-modeling approach based DSL for TFF, the TFF-DSL. This DSL decouples the task execution aspect from the coordination of the tasks in two separate DSLs. This decoupling allows the replacement of each of the DSLs to alternative formulations. Moreover, the DSL enables the integration of the behavior with different robots or software frameworks. Chapter 4 will detail the relation of the TFF-DSL with the proposed iTaSC DSL.

The Stack of Tasks [102] provides a TFF software framework, later extended to operational space. The Stack of Tasks uses a dataflow programming approach. It provides entities, which are connected manually using a scripting language. Stack of Tasks has many optimization problem solvers integrated, for resolved-velocity, resolved-acceleration, and resolved-torque problem formulations. Examples include fast solvers with regularization, prioritization, and inequality support, and QP-solvers [60, 81–83, 135].

The Whole-Body Control Framework [139, 140] provides an implementation of the operational space approach including task prioritization. It describes whole-body behavior as the composition of multiple behavioral primitives (tasks). In this framework, the different priority levels are named as constraints (primary constraints, highest priority), operational tasks (secondary constraints), and postures (tertiary constraints).

A first implementation of the iTaSC software framework [144, 147, 148] provides an Orocos component based framework. This framework distincts constraints, virtual manipulators, robots, objects, and a solver, connected to a central scene component. iTaSC application creation relies on the Orocos tooling and the interface provided by the components. The framework provides task coordination and configuration on three levels of abstraction, defined as Task-Skill-Motion.

It is the refactoring and extention of this iTaSC software framework that formed the starting point of the insights acquired in this thesis, as outlined in following section.

2.4 Timeline of developments

The outline of the software part of this thesis (Chapters 3 to 5) follows a non-chronological order of the developments. This section gives a timeline of these developments, starting from the first implementation of the iTaSC software framework.

First, the iTaSC software framework is refactored and extended. Changes include (i) hierarchical coordination of all functionality, not only tasks; (ii) scheduling of the algorithm in a correct order; (iii) including a hardware abstraction layer; (iv) a first attempt to separate the 5Cs; (v) support of robots with a branched kinematic structure; (vi) embedding iTaSC in an application; (vii) creation of deployment tooling and *boilerplate* scripts; (viii) integration with the ROS framework ecosystem; (ix) more implementations of the different component types, adding functionality to the framework. The functionality is structured in component libraries for each type of component (ROS stacks). The software follows an Object-Oriented software programming paradigm. Each component of the iTaSC framework inherits from a type template which defines the default interface.

With the growth in functionality, also the size of the scripts to create an iTaSC application grew. Moreover, the API became complex, and developers needed knowledge of many domains in robotics, and knowledge of many development languages and tools: C++, Lua, Orocos-scripting languages, Orocos-RTT, Orocos-KDL, rFSM, etc.

A first step to ease the development effort is the introduction of a DSL conforming the meta-modeling approach. As a result, one of the benefits is the effectively separation of application and implementation specific code and settings. This DSL is published in [167], of which a refactored version is discussed in Chapter 4.

Further insights in effective ways to separate the 5Cs lead to the refactoring of the iTaSC software framework following the Composition Pattern, as detailed in Chapter 5. As a consequence, the solver and the scene of the iTaSC approach looses its central role, as all models will have equal importance.

This leads on its turn to the formulation of the general principles of the Composition Pattern, detailed in Chapter 3. As a consequence, iTaSC does not

play a central role in a robotics application, but forms a possible approach to constrained-based programming in an application.

These insights are re-applied to the DSL, as detailed in Chapter 4.

2.5 Conclusions

This chapter (i) presented a broad overview of approaches that structure and implement behavior, (ii) it focussed on constraint-based programming based motion and force control, (iii) it detailed approaches to solve constrained optimization problems, (iv) and it outlined the evolution of the work that will be presented in this thesis.

The chapter showed the integration challenge posed by the large variety of functionality and hardware in robotics. The following chapters of thesis introduce the Composition Pattern methodology and software architectural pattern, to support this challenge in a uniform, constructive, and systematic way.

This thesis uses three elements of prior art described in this chapter as 'instruments'. The **5Cs principle of separation of concerns** discussed in this chapter will be a first important instrument. As a second instrument, and in contrast to many of the here presented architectures and frameworks, this thesis uses the **meta-modeling** approach to use and compose domain-specific knowledge. **Constrained optimization** forms a third instrument, it provides a methodology to composition, exemplified by the constraint-based programming approach to motion control and task specification.

Chapter 3

Systematic Robot Application Development: Applying the Composition Pattern to Constraint-Based Programming*

Robotics has seen a growth in demonstrations of complex behavior on platforms with an increasing number of degrees-of-freedom (DOF), types of actuation mechanisms, communication networks, sensors, and processors. Robot competitions among such complex, highly autonomous systems attract a lot of attention from the robotics community and beyond. Well-known examples include the Darpa Robot Challenge and the RoboCup competition. Given the scale and complexity, as well as the increasing demand for flexible application reprogramming and portability to different platforms, application development has become an effort shared by teams of developers, each with different levels and fields of expertise. In order to create an application, these developers have to create and compose compatible and interoperable building blocks. This integration process, often including parts of a team's legacy software, commonly jeopardizes the success of a project.

*This chapter is based on Vanthienen, D., De Laet, T., Bruyninckx, H. (2014), "Systematic robot application development: applying the Composition Pattern to constraint-based programming", submitted to *Robotics and Automation Magazine*.

Figure 3.1: Scene of the tomato picking running example. The PR2 robot has to pick the tomato from the counter (left), and drop it in the basket on the fridge (right). A person doing the dishes between those locations forms a dynamic obstacle during all phases of the task at hand.

This chapter addresses application development challenges through the systematic combination of *structure* and *behavior*. More concretely, it introduces the *Composition Pattern* and applies this to the development of applications that use *constraint-based programming*. Moreover, this chapter shows how to refactor existing applications to more reusable systems by looking at both these aspects in an integrated way.

Chapter 5 will describe the Composition Pattern as a *software architectural pattern*, resulting from the multiple refactoring efforts on the iTaSC software framework [166]; in contrast, this chapter first focuses on the *systematic approach to apply the Composition Pattern to the modeling of robot applications*. The approach can be applied independent of the software frameworks, tools, or languages preferred by developers.

The chapter will use a tomato pick-and-place application as running example

throughout the chapter. In this application, a PR2 robot [173] has (i) to find a tomato located in the neighbourhood of a dedicated pick-up spot, (ii) to pick up the tomato, obviously not damaging it, (iii) and to deposit it in a dedicated basket a few meters away. The platform has to operate in a cluttered and populated environment, as shown in Figure 3.1. It is evident that all these tasks should take into account the limitations of the platform, and that the whole setup conforms to safety requirements.

The running example is a typical pick-and-place robot application. It is rather 'simple' to pre-program this in an *ad-hoc manner*, provided that the robot operates in a human-shielded environment, and pick and drop location are within reach. However, when any of these limiting simplifications must be relaxed, developing the application quickly increases in complexity. Hence it becomes relevant to adopt a *systematic approach* that helps creating reusable and adaptable applications. The approach introduced in this chapter aims at helping developers to deal with this escalating complexity, which comes in many forms. For example when (i) *changing the platform*, e.g. replacing the PR2 by an autonomous humanoid robot; or adding a sensor to the PR2, and use this information where useful; (ii) *changing the tasks*, e.g. grasping the orange, instead of the tomato; grasping the tomato between the two (closed) grippers to avoid squeezing it, rather than using a single gripper (Figure 3.2); increasing the number of tasks to execute simultaneously, e.g. grasping the orange with the other gripper; or executing the tasks in a more cluttered and populated

Figure 3.2: Example grasping strategies to pick up a tomato: grasping the tomato using the gripper (left), or grasping it between the two grippers (right). The orange forms an obstacle when grasping the tomato. In an alternative scenario, the orange must be grasped, and the tomato avoided.

environment with high levels of uncertainty, e.g. when human actions obstruct the view on the tomato; or (iii) *changing the knowledge level*, e.g. replacing the given sequence of sub-tasks by a high level goal and a reasoning algorithm.

The chapter uses constraint-based optimization as a unifying approach to create robot task descriptions: every task is a set of objective functions and constraints that the robot controller has to satisfy, with contributions from joint space, Cartesian space, and/or sensor space. A major reason for the growing success of the constraint-based approach is that constraints and objective functions are *composable*. This chapter exploits this composability property by applying the constraint-based approach not just strictly to the robot tasks alone, but to all entities of a complete robotic application, such as platform-specific constraints, or constraints imposed by the manipulated object (Figure 3.5).

The chapter is organized as follows: The following, Related Work Section 3.1 links existing approaches in literature to this chapter. The next section states the Composition Pattern and describes its underlying concepts. The subsequent section applies the Composition Pattern to the example domain of constraint-based programming. Next, a discussion section details the benefits of the Composition Pattern and its role in reuse and refactoring. Further, this section compares the concepts introduced in this chapter to existing approaches. Finally, the last section states the conclusions.

3.1 Related work

This section discusses related work on constraint-based programming, and existing architectures, frameworks, and methodologies for robotics.

3.1.1 Constraint-based programming

One constraint-based programming approach, named **i**nstantaneous **Ta**sk **S**pecification using **C**onstraints (iTaSC) [46,50,51], introduces particular sets of auxiliary coordinates to express task constraints and model uncertainty. These auxiliary coordinates are specified between *object frames* defined on the robots and objects involved in the application. Where possible, these object frames have a *semantic* meaning in the context of the task, for example a specific 'corner of a table'. The *composition* of the constraints of all (sub-)tasks, defined on possibly a multitude of robots, objects, and sensors, translates to a numerical *constrained optimization problem*. The developer can introduce *weights* and/or *priorities* between the different concurrent tasks. In the instantaneous version, a *solver*

algorithm computes at each moment in time the best setpoints (for example joint velocities or accelerations) for all the robots involved in the application. A software framework and modelling tools for iTaSC are available [166,167].

Related approaches that define task specification as a constraint-based optimization problem include the Stack of Tasks (SoT) [102] framework and the Stanford Whole-Body Control framework (SWBC) [139]. The concept of constraint-based task specification and control to define the overall robot task as a composition of individual composable constraints will prove to match the composition in the Composition Pattern, as will be detailed further on. Hence it makes an apposite choice as example domain.

3.1.2 Frameworks, architectures, and methodologies

Past research resulted in different frameworks, architectures, and methodologies to deal with complexity in robotics. Kortenkamp and Simmons give an overview of robot system architectures in [95]. The following paragraphs give an overview of recent advances.

A first type of frameworks uses *hierarchical (concurrent) state machines or flow charts*, as pioneered by Nilsson [119]. Control-focused frameworks of this type include Skill/Manipulation Primitive Nets [62,155], which provide state machines of hybrid force/position control setpoints, and more recently LightRocks [156], which extends this idea using a modeling approach and introducing levels of abstraction of task specifications built on hybrid force/position control. General application-focused frameworks of this type include SMACH [21] and ROSCo [116].

Another type of frameworks starts from a *multi-tiered architecture* [22]. Angerer et al. [8] present a recent two-tiered object-oriented architecture for industrial robotics, robAPI. It consists of a robotics API tier, comprising a command and an activity layer, and a real-time robot control core tier.

The increase of knowledge and interactions between parts of knowledge results in the need to manage the represented knowledge, and the need for theories to prove properties about processed knowledge. *Knowledge driven approaches* include CRAM and temporal logic based frameworks. CRAM [16] provides a light-weight reasoning mechanism that can infer control decisions. Temporal logic based frameworks [53,96] on the other hand, synthesize a discrete plan that satisfies a (formal) high-level specification including timing constraints. Recently, Doherty et al. [54] presented a formal framework and agent-based software architecture based on delegation for automated specification, generation, and execution of high-level collaborative missions.

In contrast to the frameworks and architectures above, this chapter does not introduce the 'single best system architecture', but helps developers in defining a system architecture that fits their application's needs using a systematic approach.

Next to the frameworks and architectures mentioned above, there are a number of *framework ecosystems* that use data flow or component-based techniques. These framework ecosystems allow developers to create large systems from modular components or nodes that encapsulate certain functionality. These components are intended to be substitutable blocks of computation that communicate data or events with other components. Component-based tools possibly allow to call functions (services) on other components. Examples include modeling framework ecosystems such as LabVIEW [117] and Simulink [154], and code-oriented framework ecosystems such as ROS [171] and Orocos [33].

This section further compares different framework ecosystem aspects using the terminology of the 5Cs principle of separation of concerns [91,128], which separates the *communication, computation, coordination, configuration,* and *composition* aspects in software functionality. It forms a basis for the here introduced Composition Pattern. The PhD dissertation of Philips [127] discusses the level of compliance of different framework ecosystems with the 5Cs principle of separation of concerns. This section gives a general overview and further adds to this comparison some key differences.

In addition, this chapter considers an *entity* as a concept or model that maps to software components, agents, objects, modules, processes, activities... The framework ecosystems primarily focus on *functional entities*, conforming to algorithms or computations (data processing), and their communication. *Support entities* that 'manage' functional entities, by handling configuration, composition, coordination, monitoring, and scheduling, are of secondary importance for most of these framework ecosystems; the introduction of support entities as well as the consistency of their usage, are generally left to the programmer. In contrast, the Composition Pattern elaborated in this chapter introduces these support entities in a systematic way, already in the conceptual and architectural design phase.

Most framework ecosystems provide the possibility to separate *configurable* parameters from the computation functionality, for example using the ROS parameter server or Orocos properties. The *composition* of components is fixed by design and possibly hierarchical in the LabVIEW and Simulink case, ROS and Orocos on the other hand, allow flat but runtime changeable compositions. State machines are commonly used for *coordination*, for example rFSM [91], SMACH [21] for ROS, or Stateflow for Simulink [153]. The number of *scheduling* options varies among the tools, for example the implicit scheduling based

on block connectivity as default in Simulink, or its more advanced Common Function Call Initiators. Orocos assigns periodical (timer triggered), non-periodic (user triggered), or slave (coupled to another) activities ('threads') to components, and allows to choose a real-time scheduler or not. All of these framework ecosystems regard *monitoring* as a functionality to be created by the programmer, similar to other computations.

This chapter does not focus on the functionalities offered by these frameworks, but on the structured and systematic approach to application (architecture) design and the resulting consequences for software engineering design, which can be applied to the framework ecosystem of choice.

3.2 Composition Pattern: concepts for a systematic approach

This section defines *concepts* to divide a (robotics) problem into sub-problems. These concepts apply throughout the design, from the conceptual design to the software modelling phase.

The following subsections define the four concepts of the Composition Pattern, i.e. *metamodeling*, *composition*, *hierarchy*, and *semantic context*; the last subsection defines the Composition Pattern and its trade-offs.

3.2.1 Metamodeling

This chapter follows the meta-model approach and terminology of Model Driven Engineering [122] as advocated by Bézivin [20]. It considers *all entities to be models*, as opposed to the code-centric principle of *all entities are objects*. We restrict ourselves to the key concepts of metamodeling relevant for this chapter, and refer the reader to the work of Bézivin [20] for a discussion of the consequences of this paradigm shift.

In this chapter, a *model* captures a view or aspect of a system, it groups semantics. A *meta-model* presents the language to describe a model; it is a formal specification of an abstraction of a (sub)-domain. A model *conforms to* one or more meta-models. An *implementation* is an instance of, or is represented by a model. From a model a concrete implementation can be generated or hand-coded. However, this chapter will not elaborate on implementations; we refer the reader to Chapter 5 for a discussion on implementations.

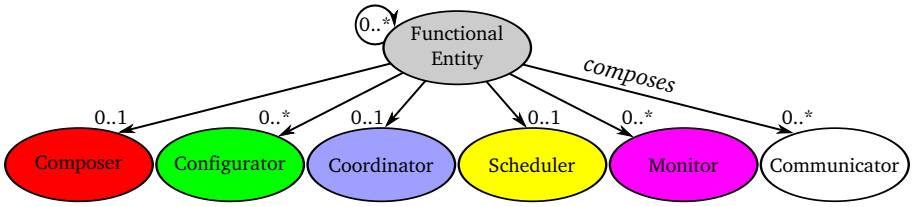


Figure 3.3: Entity types and cardinality of the entities in the Composition Pattern. Each node represents an entity of a specific type: a `Composer`, `Configurator`, `Coordinator`, `Scheduler`, `Monitor`, `Communicator`, or (Composite) `Functional Entity`. A `Functional Entity` can compose a number of these entities, indicated with the arrows and the multiplicity numbers. The dissertation names a `Functional Entity` a Composite `Functional Entity` or simply composite when it composes other `Functional Entities`. The multiplicity indications start from zero, indicating (i) that a `Functional Entity` does not need to be a composite, and (ii) that certain entities can be left out when they are not relevant for the composite at hand. The color code for each entity type will be used throughout the dissertation.

In the examples of following sections, teletype font names indicate `meta-models`[†], and italic font names indicate *models*. For example a *Tomato* `Object` denotes a *Tomato* model, conforming to the (physical) `Object` meta-model. De Laet et al. [42,43] present one example of the use and usefulness of metamodeling in robotics, compatible with the Composition Pattern. In this example a `Geometric Semantics` meta-model presents a language and rules on geometric relations and operations between rigid bodies. A concrete model represents the semantics of a specific relation or operation, for example the *End-Effector Pose* of a robot. This model can be translated to an implementation using an existing geometric library such as KDL [146] or the ROS geometry stack [66]. The metamodeling approach enables automatic checks for semantic correctness of geometric operations and representations, and their correct deduction.

3.2.2 Composition

A **Composite Functional Entity**, further referred to as 'composite', is a group of entities following a fixed pattern, shown in Figures 3.3 and 3.4. A

[†] `Teletype font names` will also be used to indicate a non-specific model that conforms to the meta-model with the same name, in places where it is clear from the context. For example a `Task` indicates a non-specific task model that conforms to the `Task` meta-model.

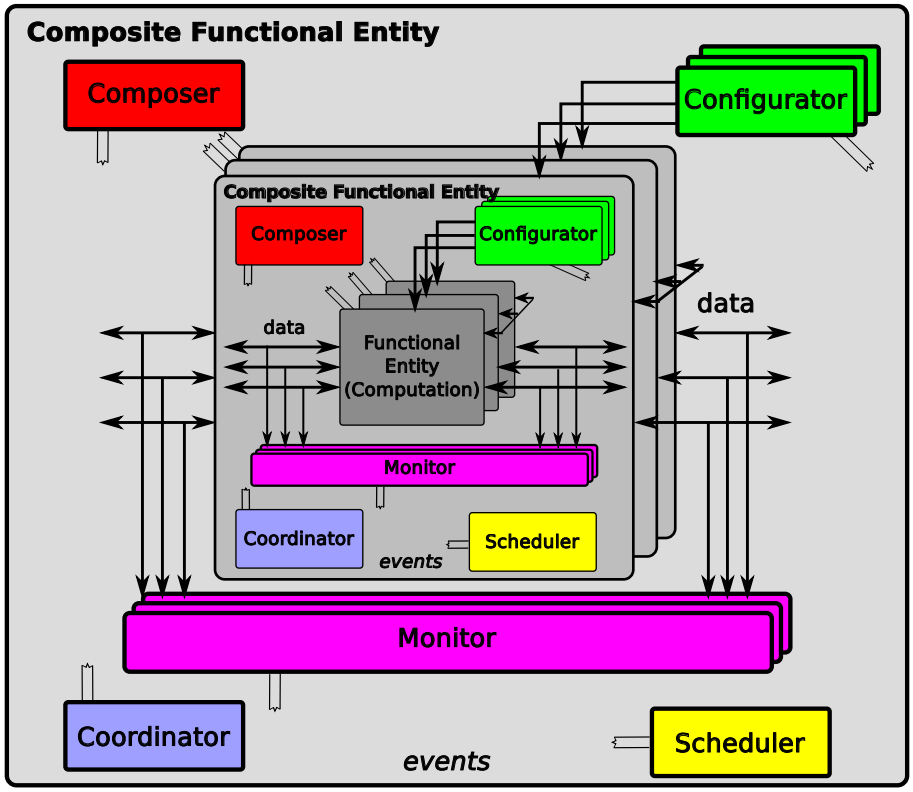


Figure 3.4: Structure of the Composition Pattern with indication of data and event communication. Each block represents an entity. The colors represent the entity type as indicated in Figure 3.3. A darker shade of grey indicates a Composite `Functional Entity` at a deeper dept level within the hierarchy. Three layered blocks indicate 'one or multiple entities of the same type', i.e. entities conforming to the same meta-model. Arrows indicate data communication and double lines indicate event communication. Since entities are broadcast, the double lines represent a 'bus system' and are only partially drawn. The figure makes abstraction of possible communication needed by the `Scheduler` to execute scheduling activities. Chapter 5 will retake and further detail this figure (Figure 5.1).

composite 'composes' entities of different types[‡], listed below, which will be

[‡]The entities are listed as *models* to be able to list their cardinality, while the examples specify models or implementations. The entity 'type' is the meta-model they conform to. This dissertation uses the listed names to represent an entity meta-model, model or instance

detailed based on the example of a `Task` composite.

- One or more **Functional Entities** (computation): A Functional Entity represents *continuous time and space behavior*, and can be in itself a Composite Functional Entity. A Functional Entity is a *'data processor'*. For example a `Setpoint Generator` and a `Controller` that tracks the trajectory from the `Setpoint Generator`, are Functional Entities that are part of the `Task` composite.

- One or more **Monitors**: A Monitor represents *conditions to verify on data flows and events to raise* based on these conditions. For example, a Monitor (implementation) that forms part of a Task could 'monitor' control errors (data from the Controller), and send out an event when a control error violates a certain condition.

- One **Coordinator**: A Coordinator represents *actions to command from the other entities within a composite.* On their turn, these entities report back to the Coordinator. It gives the composite the autonomy to handle certain situations locally. The Coordinator is an *'event processor'*, typically a finite-state machine (FSM): it receives and sends out events from and to other entities within and outside of the composite. For example, the Coordinator of the Task composite, i.e. the implementation of the *Task Coordinator*, sends out events that will trigger a reconfiguration of certain entities of the Task as a reaction on the event of the Monitor, which signaled that the control error was 'too high'.

- One **Scheduler**: A Scheduler represents *resource access and timing constraints on the different entities of a composite.* These constraints are of importance for resource allocation when an implementation is created of a certain composite. What this 'resource' is, depends on the context of the composite. For example, a Scheduler (implementation) can trigger the Functional Entities of a composite one after the other, at a constant frequency.

- One or more **Configurators**: The implementation of a Configurator *applies settings, i.e. data and parameters, to an entity of a composite* when triggered by the Coordinator. In this step the Configurator 'translates'

if this distinction is not important or clear from the context. Name typesetting or specific names are used to clarify the exact meaning where necessary. For example, the `Coordinator` meta-model specifies a language for an event processor, e.g. the rFSM DSL for finite-state machines. A `Coordinator` model represents actions to command from other entities, e.g. the concrete rFSM model for a certain task. A Coordinator implementation raises events to command other entities when it receives events from a Monitor, e.g. an implementation of the rFSM model of the task which receives a 'limit reached' event, changes the internal state of the Coordinator, and sends out a 'stop execution' event.

events from a coordinator to the data and parameters in the context of the entity. Therefore, the Configurator is a 'parameter translator' and the point where knowledge from a knowledge base can be introduced. Klotzbücher et al. introduced this separation of commanding and executing configuration as the Coordinator-Configurator pattern [89]. For example, the Coordinator of the Task composite can, as a reaction to the 'too high control error' event from the Monitor, command a Configurator to reconfigure the Controller.

- One **Composer**: A Composer represents how *all entities within composite are grouped and connected*, it is a model of the *architecture* of the composite. For example the `Composer` states how the `Setpoint Generator` can be composed or connected with the `Controller`.
- One or more **Communicators**:[§] A Communicator *models constraints on how entities exchange data and events* over a certain connection. Bi-directional data communication occurs between Functional Entities within and outside a composite, as will be detailed in the following section. Monitors monitor the data flow of Functional Entities of a composite, hence communicate with them. Configurators communicate with the other entities within a composite to set parameters. All entities send and receive events. Events are broadcast, also outside the boundaries of the composite. For example the `Communicator` models that a buffered connection is needed to receive events that the *Task Coordinator* receives from the `Monitor`.

As hinted at in Section 3.1, the Coordinator, Composer, Scheduler, Monitor(s), and Configurator(s) are refered to as the *support entities*. Of the support entities, the Coordinator, Composer, and Scheduler are singletons within a composite since they take decisions for the whole composite; in contrast, multiple Configurators and Monitors can exist –and be executed– in parallel since they can have a smaller scope. It is however possible that not all concerns are relevant for the composite at hand, and hence certain entities can be left out. For example the Monitor can be left out, when the composite has no data to monitor. The `Task` example of a composite will be detailed in more detail further on in Section 3.3.3 and Box 3.3.4.

Typically, a developer is not confronted with the separated entities which he can compose, but with a set of functionalities to model or implement. The developer has to separate this set of functionalities in entities, inspired by

[§]Communication and scheduling are well studied topics. Although important concerns, this dissertation did not focus on communication or scheduling models in detail. Since the experimental validation only used single robot platforms, communication did not pose a limitation to the execution and performance of the researched applications.

the 5C's approach to separation of concerns, before composing the separated entities to a (hierarchy of) Composite Functional Entities, as described above. Section 3.4.2 will give guidelines on how to use the Composition Pattern to refactor existing applications.

3.2.3 Hierarchy

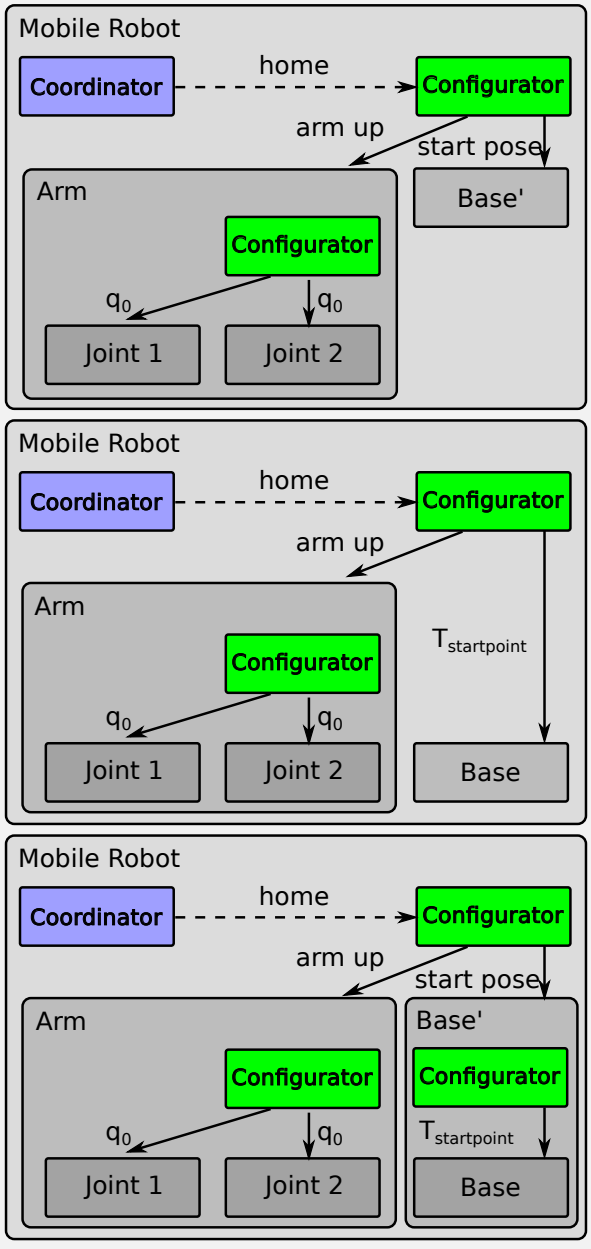
The Composition Pattern helps to derive a set of modular entities as building blocks that are easily adapted or replaced, since each entity's behavior has a limited scope (separation of concerns), and a clear meaning. Applying the composition pattern, results in a tree of entities with a recurring, fractal structure. The level of granularity of the leaf nodes of the composition tree, i.e. their 'depth', does not need to be identical for all branches of the tree. Hence each Functional Entity can be replaced by a composite, until the granularity required by the specific application is achieved. At a design phase, a trade-off needs to be made based on considerations such as the existing functionality at your disposal and efficiency. In a refactoring phase, these levels can change, allowing for an incremental evolution of the application. Remark that as a consequence of the tree of composition all Functional Entities (computations) are always leaf entities.

Although the composition is strictly hierarchical, the 'communication' (fifth C of the 5Cs principle of separation of concerns) does not need to be hierarchical. Functional Entities communicate data on the same level of (compatible) semantics, although they may reside on different depth levels, therefore crossing different composition boundaries. The common parent entity checks and connects communication channels. For example a *controller* entity communicates setpoints to the *joint1*, *joint2*, and *base* or *base'* entities presented in Box 1. As a consequence a flat or a hierarchical composition are similar from the perspective of data-flow between Functional Entities, since data is not bound to the limits of a composite.

3.2.4 Semantic context

Every composite forms a **semantic context**; i.e. the entities within a composite use a shared vocabulary. The support entities translate from the context of a composition to the context of its child functional entities. This concept is important for knowledge driven architectures, where this context needs to be explicit. Box 1 gives an example of the relation between the *Mobile Robot*, *Base*, and *Joint* contexts. In the running example, higher level compositions use tomato-specific semantics, such as 'a rotten tomato' while the lower level

Box 1: Example composition and inter-context translation



The figures above consider the example in which the *home* position should be configured on a mobile robot consisting of a mobile base and an arm. A *Mobile Robot* Composite `Functional Entity` composes a *Base'* or *Base* `Functional Entity`, and an *Arm* Composite `Functional Entity`. The latter *Arm* entity is common to the three figures, it composes the $Joint_i$ `Functional Entity`. Hence, the $Joint_i$ entities are at depth two (D2), and the *Arm* entity at depth one (D1) with respect to the root composite *Mobile Robot* at depth zero (D0). The leaf nodes (functional entities) control the base as a whole (*Base'* or *Base* entity), and each joint of the arm respectively ($Joint_i$ entities). In this example the three depth levels coincide with different levels of abstraction. Entities at a deeper depth level have a darker shade of grey. We consider three variations of this example, as shown from left to right:

1. In the figure on top, the D0 *Mobile Robot Configurator* configures *Base'* and *Arm* with parameters ('arm up', 'start pose') that belong to the same level of abstraction. The *Base'* is able to interpret and act on parameters of this level of abstraction. The support entities of the *Arm* composite further translate this parameter to the concrete numerical joint setpoints q_0.

2. In the figure in the middle, *Base* is an entity that can only interpret parameters of a lower level of abstraction, similar to the level of abstraction of the $Joint_i$ entities. The D0 *Mobile Robot Configurator* is adapted with respect to the top figure, to translate the 'home' event to two parameters 'arm up' and '$T_{startpoint}$', a concrete pose of the base. However, the need for adaption of the D0 *Mobile Robot Configurator*, as well as the translation to different abstraction level is regrettable from a design point of view.

3. In the figure on the bottom, the D2 *Base* entity is wrapped by a composite *Base'* that translates the 'start pose' event received from the D0 *Mobile Robot Configurator* to '$T_{startpoint}$', which *Base* can interpret. This case represents one possible way to integrate the existing *Base* entity in the left figure composite, while preventing duplication of parts of models or code, such as the D0 *Mobile Robot Configurator*.

The first two examples demand less effort to develop. However, in the long term, the third option, consisting of fine-grained entities with depth levels coinciding with levels of abstraction, will prove to be more reusable.

composition that generates robot motions, uses robot-specific semantics, such as 'joints'.
The semantic context also forms a 'boundary'. The support entities of a composite 'know' only about the other entities within that composite, not the composition of the parent or child Functional Entities. Moreover, the Functional Entities of a composite do not know about the support entities that *manage* them, they send and receive data and events not knowing who will use or react on them. It does not imply information hiding however: child entities can be introspected or reasoned about.

3.2.5 Definition of the Composition Pattern

The Composition Pattern is an architectural pattern to structure, i.e. to (i) contain and (ii) connect, (iii) types of behavior, i.e. the different entities as listed in Section 3.2.2 and shown in Figure 3.3. The pattern allows following trade-offs for each of the three:

(i) The definition of the semantic context of a composite, and in addition the depth level in the composition tree and level of abstraction to which functionality is modeled: Therefore, how much and in which detail is that part of the application modeled? For example, robots will be modeled as ideal 'velocity following devices' in the examples of this dissertation.

(ii) The level of communication allowed trough the boundaries of the composite, as shown in Figure 3.4.

(iii) The cardinality of the different entities within a composite following the constraints shown in Figure 3.3. For example:

- What parts of the coordinators in the (composite) functional entities can be replaced by the higher level coordinator?
- How many Functional Entities does a Monitor monitor?

The trade-offs are limited through the constraints of the pattern that need to be followed. For example, there can be maximal one Coordinator for each composite. Other constraints in the pattern are the partial ordering of the composer, coordinator and scheduler: the composer influences the behavior slower than the coordinator, which on its turn influences the behavior slower than the scheduler.

The following section details the application of the presented approach to the domain of constraint-based programming.

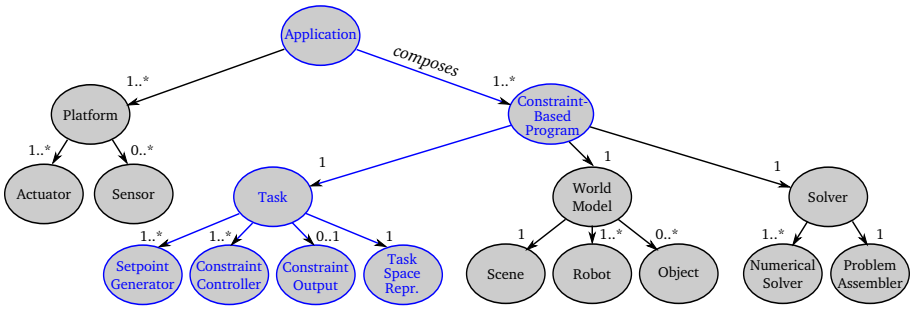


Figure 3.5: Composition tree of a robotic constraint-based application. A node represents an entity meta-model. An arrow indicates composition: the node at the beginning of the arrow composes entities that conform to the meta-model at the end of the arrow, with a multiplicity indicated next to the arrow. Moreover, a composite functional entity can compose entities of the same meta-model, which is not shown on the figure. For example an entity conforming to the `Task` meta-model can compose different entities that conform also to the `Task` meta-model. The application shown on top is the root composite, the entities shown at the bottom are the leaf entities considered here. The text will focus on the branch of the tree shown in blue.

3.3 Applying the Composition Pattern to constraint-based programming

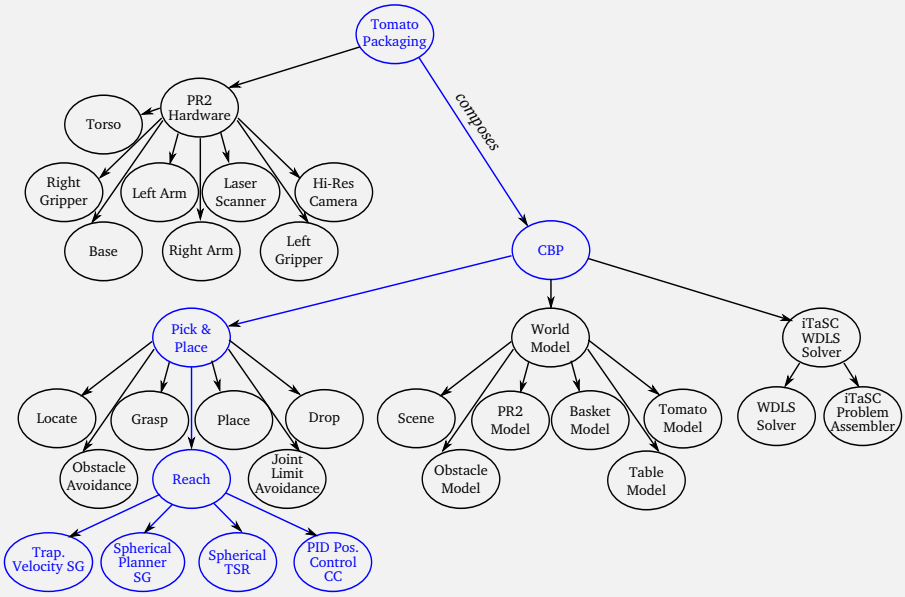
This section explains how robots applications can use the concepts of structured application development introduced in previous section. It describes a generic division of the domain of constraint-based programming in a composition tree.

Figure 3.5 gives an overview of the composition tree of a constraint-based application. Each node of the shown tree represents a meta-model of a (composite) functional entity. On the one hand, it shows the relation between meta-models. On the other hand, it shows the hierarchy of composite functional entities, complementary to the composition as shown in Figure 3.3 (each node has the structure as shown in Figure 3.3). This section focuses on one branch of this tree, shown in blue, choosing a limited number of entities of a composition to detail further on.

Furthermore this section applies the division represented by this tree to the running example, resulting in concrete models shown in Box 2.

We define following (composite) entities within the task specification branch of the application tree, from root to leaf: `Application`, `Constraint-Based`

Box 2: Composition structure of the running example



The figure above presents the composition tree for the running example, using the same notation as in Figure 3.5 with the difference that a node is a (composite) functional entity *model*. It forms one of the possible models for the problem at hand.

The composition tree examplified above does not present the only possible hierarchy. First, the depth of the composition tree does not need to be restricted. For example a `Setpoint Generator` generating position setpoints can be implemented or replaced by a set of single DOF trajectory generators. In this case, the `Coordinator` of the now composite `Setpoint Generator` entity manages the different possible timings between the six child `Setpoint Generators`.

Moreover, intermediate composition layers can be introduced. For example an intermediate layer can be introduced between the *Pick and Place* `Task` and the different sub-`Tasks` (not shown in the figure). More concretely, the *Reach* `Task` could be replaced by a composition of the current *Reach* `Task` with an *Arm Guide* `Task`. The latter constrains the arm to move within a tight subspace when grasping a tomato in a hard-to-reach location.

Even more, entities can have multiple roles. For example the `Setpoint Generators` considered in the running example (*Trapezoidal Velocity SG* and *Spherical Planner SG*) deliver a fixed setpoint or a time-dependent stream of setpoints deduced from a motion profile. These `Setpoint Generators` get their goal from another entity, such as the *Configurator* of the *Pick and Place* `Task`, or a *Planner* (also an instance of a `Setpoint Generator`) outside or inside the scope of the *Reach* `Task`. However a `Setpoint Generator`, delivering setpoints to the `Constraint-Controller` can be of a different form, or defined outside of the scope of the `Task`. For example the haptic teleoperation scheme using iTaSC introduced by Borghesan et al. [24]. In this scheme the `Constraint-Output` of the position-coupling `Task` at the master side forms the `Setpoint Generator` of the equivalent `Task` at the slave side, and vice versa.

Program, and Task.

3.3.1 Application

An Application attaches a Constraint-Based Program to specific Platforms (hardware resources, which can be virtual for simulation). These Platforms consist of the specific robot Actuators, i.e. motion capabilities, and Sensors, i.e. sensing capabilities.

The running example will make use of the *PR2* and the default sensors of the platform: the tilting laser scanner and (stereo) cameras on the head.

3.3.2 Constraint-Based Program

A Constraint-Based Program (CBP) defines task specification and control on a robot setup. It composes a Task and attaches the Task to the world model, at certain points where we define *object frames*. The Constraint-Based Program also comprises a Solver that computes the control input to the robot as a solution of the constrained optimization problem. The World Model consists of the Robot- and Object (kinematic and dynamic) models placed in the Scene.

The Constraint-Based Program of the running example composes following models: (i) *Pick and Place* Task, (ii) a *World Model*, (iii) and a *Weighted-Damped Least-Squares Solver*. The *World Model* composes a *Scene*, a *PR2 Model*, conforming to the Robot meta-model, and models that conform to the Object meta-model: one or more *Obstacles* to avoid, the *Table* where to pick up the tomato, the *Basket* where to put the tomato, and the *Tomato*.

3.3.3 Task

A Task can compose different sub-Tasks, in which case an intermediate composition level is introduced. To make the distinction we will define a *Composite Task* composing different Tasks. The *Composite Task Coordinator* coordinates the active set of tasks. It commands different global weights and priorities as well as (abstract) goals for the tasks.

The *Pick and Place (Composite)* Task of the running example can be implemented using different combinations of tasks. The *Composite Task* developer chooses these Task entities, by (re-)using existing Tasks from a

library, or by asking a `Task` developer to develop a model and implementation that fits the purpose. For the running example, he chooses to model the desired behavior using following `Tasks`:

- a *Locate* `Task` to actively look for the tomato, defined between the tomato-sensor hardware (a camera or laserscanner for example) and the pick-up spot,
- a *Reach* `Task` to reach for the tomato once found, defined between the tomato and a gripper,
- a *Grasp* `Task` to grasp the tomato, also defined between the tomato and a gripper,
- a *Place* `Task` to position the tomato in the basket, defined between the tomato and the basket,
- a *Drop* `Task` to simply release the tomato, defined on the gripper,
- platform related safety `Tasks` such as *Joint Limit Avoidance*, defined on the joints of the robot (configuration space),
- and *obstacle avoidance* `Tasks`, defined between obstacles and robot parts.

Some of these tasks will be executed sequentially (locate - reach - grasp - position - drop), others in parallel. The *Pick and Place Coordinator* decides on this behavior. Remark that the resultant robot behavior emerges from the composition of the constraints of all active `Tasks`, for example a simple *Reach* `Task` will only be successful if combined with the necessary safety and obstacle avoidance tasks.

A single `Task` consists of a set of constraints on a task space representation[¶]. It is however unaware of its concrete purpose within an application. Even the object frames in between which the `Task` is defined are unknown to the `Task`. It is the parent of the `Task` that defines its purpose by coordinating, configuring, scheduling, monitoring, and composing it.

For example the *Reach* `Task` of the running example composes alignment constraints to align the gripper with the vector between the object frames in between which the *Reach* `Task` is defined, and an approach constraint that reduces the distance between these object frames. It is the *Pick and Place* `Task` that defines the object frames to be on the *Tomato* and the *Gripper* models.

At a meta-model level, a `Task` composes:

[¶]Section 2.2.2 defines constraint-based programming and iTaSC related terminology.

- a `Task Space Representation` (`TSR`), which defines a representation of the task space, e.g. a spherical coordinate system for the *Reach* `Task`;
- a formulation of a `Constraint-Output` equation (`CO`), which defines the output as a function of the state of the `TSR` and the joint space (*joint coordinates*), e.g. the selection of the spherical coordinates of the `TSR` forms the `CO` for the *Reach* `Task`;
- one or more `Constraint-Controllers` (`CC`), which define the controller on the output, e.g. a position controller imposes the constraints for the *Reach* `Task`;
- and one or more `Setpoint Generators` (`SG`), which define the desired values of the output at each time instance, e.g. an interpolator and a planning algorithm deliver the setpoints for the `CC` of the *Reach* `Task`.

In the running example, the *Coordinator* of the *Reach* `Task` decides when to switch between the two provided `Setpoint Generators`. Box 3 details the interaction of the *Reach* `Task` with its parent and leaf entities. Other examples of `Tasks` of the running example include a set of inequality constraints for each joint of the platform, which implements the *Joint Limit Avoidance* `Task`, and a simple open-close algorithm monitoring a 'touch' condition, which implements the *Grasp* `Task`.

The presented composition stimulates developers to make all assumptions on safety or platform specific constraints explicit, structured following Figure 3.5. Safety and platform specific constraints such as joint limit avoidance, center of mass requirements for humanoids etc. are introduced as `Tasks`, since they constrain the robot platform in the same way as any other `Task`. Making these `Tasks` explicit, allows human or artificial reasoning on the full active set of `Tasks`, without the need for discovering hidden assumptions. In the simplest case, the developer has to define these `Tasks` himself. However, tooling can add these safety and platform-specific `Tasks` automatically, based on the selected platforms.

3.3.4 Remarks

The tree of composition is a basic blue print for applications using constraint-based programming, it is a policy to use the Composition Pattern that gives definitions to guide decisions and achieve rational outcomes. However some level of flexibility remains as detailed in Box 2. Abovementioned Subsection 3.3.3 gives an example where an intermediate *Composite Task* level of composition is introduced. Moreover, the here described tree is not intended to be 'complete':

entities not mentioned in the division can make part of a composite, for example `Estimators`.

Although presented in a top-down order for readability, the typical workflow will start at an intermediate composition level, composing existing (composite) entities from libraries. For the running example, the applied workflow was (i) first the development and choice of the `World Model` and `Solver` within a `Constraint-Based Program`, (ii) second the development of the *Composite Task* by selecting the `Tasks` and their interactions, (iii) and last the embedment in hardware, creating the `Application`.

Remark that each composite or functional entity gives rise to a different user perspective at a certain level of abstraction, demanding a different (level of) expertise.

Further remark that the different entities in the running example can be applied more generally, outside the scope of tomato picking. For example the approach to tomato picking can be generally applied to ball-shaped objects. However, naming of entities and events of the running example are kept within the scope of the example for readability. The following section will discuss this generality.

3.4 Discussion

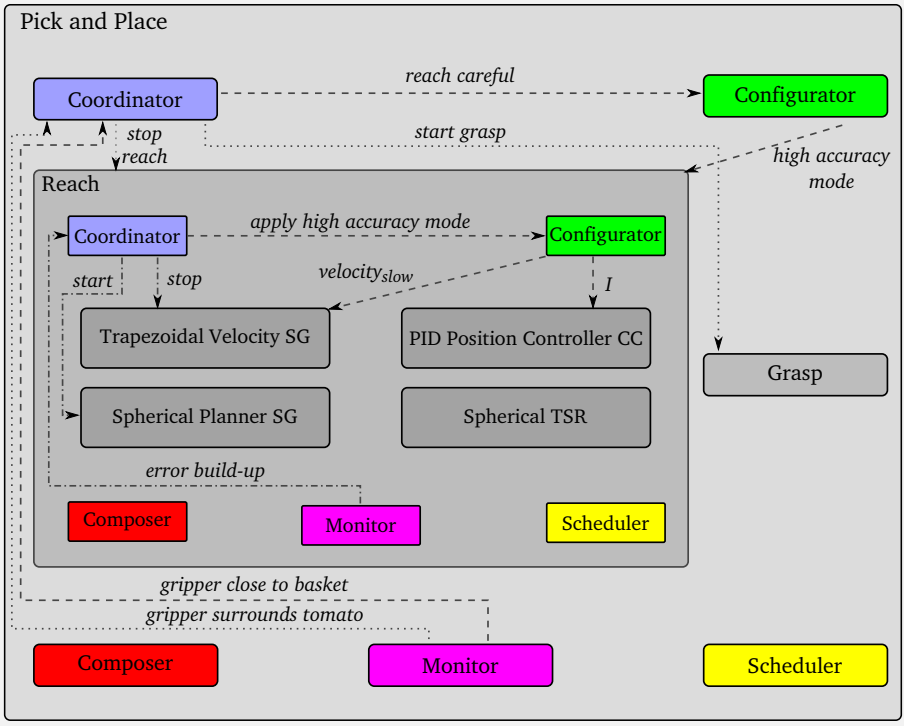
This section first discusses the implications and benefits of the Composition Pattern, secondly it gives guidelines to apply the Composition Pattern, lastly it discusses the relation to existing frameworks, architectures, and methodologies.

3.4.1 Implications and benefits of the Composition Pattern

The Composition Pattern helps the application developer to avoid following 'bad design' traits [103]: *rigidity*, i.e. when every change has its effect on too many parts of the system, *fragility*, i.e. when a change breaks unexpected parts of the system, and *immobility*, i.e. when the reuse of a piece of the system is hard since it is entangled with the application it was first designed for. These three 'bad design' traits characterize their respective opposites: flexibility, robustness, and reusability. We interpret these traits not only as a static architecture problem, but also as a dynamic, run-time, and behavior problem.

The Composition Pattern decreases **rigidity** since it *separates concerns* and *specifics are filled in as late as possible*. For example, configuration models what parameters to change and how to change them, avoiding coupling to

Box 3: Example entity interaction



The figure above details the interaction of the *Reach* `Task` with its parent, i.e. *Pick and Place*, and children, i.e. *Trapezoidal Velocity Profile* `Setpoint Generator`, *Spherical Planner SG*, *PID Position Controller CC*, and *Spherical TSR*. The figure shows entities at a deeper depth level in a darker shade of grey.

The following paragraphs elaborate the different interactions, starting from the interactions of the *Pick and Place Coordinator* with the different entities of the *Pick and Place* composite. Remark that in order to interact, each composite needs common vocabulary, which differs from the other composites. The support entities translate between the composite's own vocabulary to the vocabulary of their child (composite) functional entities, as will be exemplified in following paragraphs.

The responsability of a `Coordinator` is to change behavior by interpreting and reacting on events. First the `Coordinator` coordinates the deployment

of the composite when triggered by its parent. For example, the *Pick and Place Coordinator* orders the *Pick and Place Composer* to interconnect the different `Tasks` and support entities, and the *Pick and Place Configurator* to load initial configurations to all entities, including all support entities. For example, the *Pick and Place Scheduler* is configured to schedule all concurrent `Tasks` in parallel.

Further the `Coordinator` interprets and reacts on events within the composite. For example, the *Pick and Place Monitor* monitors and signals events such as the *gripper close to Basket* condition, which triggers the *Pick and Place Coordinator* to demand the *high accuracy mode* explained in the following paragraph. The same *Pick and Place Monitor* signals *gripper surrounds tomato*, which triggers the *Pick and Place Coordinator* to transition to the *Grasp* `Task`. The dotted arrows indicate the latter. Remark that a `Monitor` signals the (non-) violation of a condition, not the expected reaction on that condition.

Furthermore the `Coordinator` triggers configuration. For example in the running example, the *Pick and Place Coordinator* commands to *reach careful when close to Basket*. The *Pick and Place Configurator* translates this command to a *high accuracy mode* configuration of the *Reach* `Task`. The *Reach Coordinator* and *Configurator* translate on their turn this mode to a lower approach speed configuration of the *Trapezoidal Velocity Profile* `Setpoint Generator`, and the activation of an integral term in the *PID Position Control*. The dashed arrows indicate these events.

The concrete translation values a `Configurator` uses, i.e. *configuration of the* `Configurator`, can be provided by different sources, including loading simple parameter lists or querying and reasoning on knowledge databases. For example, the speed configuration will depend on the controller type and the platform used.

Further a `Coordinator` has local responsibility more than the above presented translation of commands from higher levels. For example, in the running example, the *Reach Coordinator* switches from the *Trapezoidal Velocity Profile* `Setpoint Generator` to the *Spherical Planner* when the *Reach Monitor* signals that there is stall in the progress towards the goal indicated by a build-up in position error. The dash-dotted arrows indicate these signals.

other aspects, such as coordination or algorithms in functional entities. As an illustration, changing the execution rate of the running example at runtime will influence but should not alter any of its `Setpoint Generator` functional entities. Moreover, monitoring is important to decrease rigidity, since it allows an application to react on internal and external changes.

Further the Composition Pattern decreases **fragility**, because it makes the semantic *context* explicit, which keeps impact of changes *local* to a subpart of the composition tree. For example, monitoring boundary conditions, reacting (locally) on the violation of these conditions, and possibility querrying an external database for a solution in the local context, decrease fragility. As an illustration, disabling robot base movement in the running example, while the robot has to reach for the tomato outside the workspace of its arms, will cause the *Reach Monitor* to signal that no progress is made towards the goal, and the *PR2 Left Arm Monitor* to signal a stretched arm condition. The *Pick and Place Coordinator* can freeze the task execution, and signal this event. The latter will on its turn trigger operator interaction, or simply a re-activation of the base.

Moreover the Composition Pattern decreases **immobility**, because of the limited scope of a semantic *context*, the *granularity* of the composition tree hierarchy, the *modelling approach* and the *separation of concerns*. It captures a certain view of a system, making abstraction of implementation details. For example, a (composite) functional entity does not know its purpose, connectivity, or meaning within the application. Further an appropriate depth provides elementary entities, specific enough to be easily translated to code. As an illustration, the *Trapezoidal Velocity Profile* `Setpoint Generator` used in the *Reach* `Task` of the running example can be easily reused in the *Place* `Task` since its management and configuration are decoupled from its functionality. Moreover, not only leaf entities can be reused, for example the *Reach* `Task` on its own can be reused in another `Constraint-Based Program`.

3.4.2 Guidelines to use the Composition Pattern

The running example elaborated throughout the chapter specifies only one possible pick-and-place robot application. Developing this example using the Composition Pattern requires more effort than an ad-hoc approach. However, common situations include the need to extend this application to multiple consecutive and/or concurrent tasks, or port this task to another robot platform or object to manipulate. Applying the Composition Pattern reduces the effort of extending and revising the running example on a longer term. This need for extensibility, reusability, and adaptability forms one of the drivers to refactor robot application software. This section discusses some guidelines and examples

to create new or refactor existing applications to more reusable and adaptable systems.

Important is to **consider everything a model**, as advocated by model driven engineering, such that applications can be first modeled, analyzed and verified abstractly before code is generated, next to the advantages of conceptual simplicity, high scalability, and good flexibility [20].

A developer should perform a domain analysis in order to construct a meta-model. He or she should identify the domain and gather knowledge of it. This knowledge includes the domain terminology, concepts, and interdependencies and variabilities between these concepts [107]. A developer should detect the different forms of knowledge and expertise, and **divide** the application domain at hand: (i) in levels of abstraction, making general applicable sub-parts explicit, (ii) and the different forms of knowledge and expertise. This division results in the hierarchy (tree) of semantic contexts, where the developer makes the trade-offs on containment and connection (Section 3.2.5). An appropriate depth of the tree provides elementary entities specific enough for the available tools to be translated to code. It is a trade-off between *composability*, i.e. how easily an entity can be reused, and *compositionality*, i.e. the predictability and performance of behavior of a composite knowing the behavior of its components [32]. For example, as mentioned in previous section, the models used in the running example are applicable to a wider scope of problems than tomato pick and place applications. Many parts of the *Tomato Constraint-Based Program*, such as the *Pick and Place* model, can be reused to handle for example oranges: only the *Tomato* model should be replaced by a *Orange* model, as do perception algorithms. Context dependent configuration parameters can be deduced from this altered model: the force used to grip the orange, the condition for successful approach, etc.

The Composition Pattern does not replace the domain analysis phase, i.e. it does not help to determine the concepts of a domain, but aids to separate the concepts of the domain in concerns and to compose them. Mernik et al. [107] gives an overview of different approaches for each step of DSL design, including domain analysis.

Each of the semantic contexts can be **separated in concerns**, more concretely the different entities of a Composition Pattern composite. The following questions help the developer to apply this separation and make the trade-offs on behavior (Section 3.2.5) for a given semantic context (or a piece of code that implements it):

- What is the core *behavior* of the composition? The answer determines one or more **Functional Entities** (computation) of a composition. It is

the first question a developer should ask himself, since following questions detail how this behavior is 'managed'. In existing code, move only the (re)action of if/else statements, i.e. the consequent or its alternatives, to one or more Functional Entities.

- What conditions of this composite need to be *monitored*? The answer determines one or more **Monitor** entities. In existing code, move the condition of if/else statements to a Monitor.
- How to *coordinate* the behavior of this group of entities? The answer determines the **Coordinator** entity. In existing code, move the selection of the reaction to the condition of if/else statements to the Coordinator.
- Which *timing constraints* between the entities are important? The answer determines the **Scheduler** entity.
- How to apply *configuration* to the entities? The answer determines the **Configurator** entity. Replace magic numbers in existing code with configurable parameters, and move the concrete numbers to configuration
- How are the entities *interconnected* within the composite? The answer determines the **Composer** entity. If part of existing code makes assumptions on *where* the data comes from, move this dependency (where) to the Composer.
- Which constraints are important for each connection? The answer determines the **Communicator** entities.

Remark that proper design is more than separation: each entity should be adaptable. For example Functional Entities should contain adaptable behavior. However, few algorithms are ready for this level of adaptability. For example the *Trapezoidal Velocity Profile* `Setpoint Generator` in the running example is a functional entity, which algorithm can be implemented in different ways. Depending on the underlying algorithm, it can assume a fixed rate of operation (sample time), can handle a change in rate of operation, or can handle asynchronous triggers (event-driven). The latter offers a higher level of adaptability.

Chapter 4 will detail use cases using the composition tree (meta-model) of constraint-based programming, for example the use case of replacing the robot in an application with another one. Chapter 5 will detail guidelines as lessons learned from the application of the Composition Pattern as architectural pattern to the iTaSC software framework.

3.4.3 Relation to existing architectures

The Composition Pattern generalizes concepts to which existing frameworks and architectures conform to a greater or lesser extent. In the first place the Composition Pattern delivers *structure*, not making claims on the actual behavior, nor limiting the structure to a number of tiers or levels. It provides a way to improve or create applications using for example the framework ecosystems mentioned in the Related Work section. Moreover, it provides structure beyond (coupled) coordination-configuration using hierarchical state machines or flow charts frameworks.

The Composition Pattern stimulates context structuring, but does not impose information hiding. It presents a (composite) functional entity as first-class entity, which can be inspected and reasoned upon, compatible with knowledge-driven approaches such as CRAM. Moreover, in future applications we want to integrate reasoning on all composites, on all tiers, as presented in the Composition Pattern, and in contrast to 2- or 3-Tier architectures of the Related Work section.

One consequence, together with the non-strict hierarchical communication, is local reaction on events. A powerful feature, which needs to be used wisely to avoid immobility and fragility. For example a 'motor broken' event can trigger the immediate deactivation of a task, without the need to trickle through hierarchical layers, such as in strictly hierarchical architectures e.g. JAUS [78].

Remark that the Composition Pattern applied to constraint-based programming resulted in a hierarchy of entities. Other task specification approaches use different forms of hierarchy. Certain frameworks, such as TaskNets [155], use hierarchies of a single type of entity, comparable to the relation of the `Task` entities and the *Composite Task* in the running example. Other frameworks, such as the High-Level Mission Specification [54], transform high level descriptions to low level descriptions, i.e. a reduction of system complexity through abstraction along the task dimension. However, the here presented hierarchy corresponds to higher levels of platform coupling, next to the task dimension within the task tree: the application level couples the 'abstract' program to a specific hardware, while a task is the abstraction of a set of constraints. Hence the hierarchy of the high-level mission is complementary with the here presented approach, and is topic of ongoing research.

3.5 Conclusions

This chapter introduces the Composition Pattern to aid systematic robot application development (integrating structure and behavior). It does not limit itself to only 'bringing functionality together', but adds the important application design concepts of (i) metamodeling, (ii) composition (Coordinator, Composer, Configurator, Scheduler, Communicator, and Monitor), (iii) hierarchy, and (iv) semantic context.

As strongest point, the Composition Pattern aids developers to deal with the increasing scale and complexity of robotic applications, as well as the resulting need for flexible, reusable, and adaptable software. Moreover, the approach can be applied to the developers' framework ecosystem of choice. However, the Composition Pattern is not formalized in computer readable models (yet) with which tools can be created to help human developers, and in a later stage reasoning can be applied by the robots themselves at run-time.

The chapter states how the methodology decreases the design pitfalls of rigidity, fragility, and immobility and gives guidelines to develop from scratch or to reuse and refactor existing designs. However, it does not –and can not– provide the final answer on *how* to best apply the approach to any new application, because of the high impact of the structure and behavior of a specific domain.

Hence a lot more work is required to provide a broader set of structural and behavioral *models* within the robotics community, and the development of *tooling* to aid developers at creating applications; a wide and consistent application of the Composition Pattern might be a significant driver to accelerate these developments.

Chapter 4

Rapid application development of constraint-based task modelling and execution using domain-specific languages*

4.1 Abstract

Current state-of-the-art robot program development needs expert programmers. Moreover, most robot programs developed today are robot hardware and software specific, and therefore little reusable without modifications. This chapter realizes easier robot (re-)programming, by software framework independent models that can be executed using different hard- and software platforms. First, the chapter focuses on the formalization of the tasks to be fulfilled by a robot, more specifically constraint-based programming tasks using a domain-specific language (DSL). Second, it gives a reference implementation

*This chapter is partially based on Vanthienen, D., Klotzbücher, M., De Schutter, J., De Laet, T., Bruyninckx, H. (2013), "Rapid application development of constrained-based task modelling and execution using Domain Specific Languages", *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems.*, Tokyo, 3-8 November 2013 (pp. 1860-1866).

in Lua [74]. The presented DSL makes it easier to develop applications, yet is powerful to execute. It enables automatic model verification and code generation for different hard- and software platforms, diminishing code debugging efforts. Experimental validation shows the ease of creating an application and adapting it, the reduction of the amount of hand-written code, and the debugging aid offered through meaningful errors returned by model verification.

4.2 Introduction

You have an important demonstration to give on your robot, and as Murphy predicted, the robot breaks right before your demonstration. If you only could quickly change to the other robot in the lab, which unfortunately has another kinematic structure. Of course you'll have to adapt your tasks to the new kinematic structure, with another number of degrees-of-freedom, adapt your control gains, redefine tasks, reconnect and configure all parts of the code... Or don't you? If the task concept and software were separated from your platform description, your problem would be easier to solve. The example outlines the motivation for this work: simpler robot (re-)programming, by software framework independent models that can be executed using different hard- and software platforms.

Different languages have been developed to model and separate concerns involved in a robotic application. Simmons et al. [143] introduced a Task Description Language for robot control, generating a high-level task tree. Nordmann et al. [121] introduced a domain-specific language (DSL) for rich motor skill architectures and automated code-generation from the model. Ingés-Romero et al. [75] on the other hand focused on a DSL to express run-time variability, using an optimization problem to bind variability at run-time. These approaches focus primarily on the 'higher-level' task descriptions and scheduling, but have rather generic domain models for robot control tasks. This chapter however focuses on the formalisation of the tasks to be fulfilled by a robot, more specifically constraint-based programming tasks. Furthermore, it gives a reference implementation in Lua [74].

Constraint-based programming imposes constraints on the modeled relative motions between robots and objects. The chapter introduces a DSL that formalizes and structures constraint-based programming applications in robotics, in a way that is simple to use, yet powerful to execute. It further separates concerns, enabling a platform- and application-independent model, and enables automatic model verification and code generation. However, the proposed DSL does not describe all sub-domains of an application, but permits the integration

of more specific DSLs such as rFSM [94] for finite-state machines. Hence it forms a DSL between 'higher'-level domains, such as symbolic reasoning or planning and 'lower'-level domains such as control.

DSLs have great potential within robotics to aid robot programming by formalizing domains and enabling automatic model verification and code generation. The Geometric Relation Semantics [41–43] project is an example of such a DSL in robotics that shows the assistance of modelling in robot programming. It focuses on the formalization of a small domain and delivers tooling for easy use and integration.

This work uses the instantaneous Task Specification and estimation using Constraints (iTaSC) framework [46], a generalization of constraint-based programming (CBP) that uses particular sets of auxiliary coordinates to express task constraints and model geometric uncertainty. iTaSC describes a robot application as an optimization problem consisting of a set of constraints and one or multiple objective functions. A software implementation of this framework [165,166] is available under an open-source license. The framework can handle any kind of robot that can be represented as a kinematic tree.

This chapter follows the meta-model approach of Model Driven Engineering (MDE) [122], introducing the concept of domain-specific languages (DSL) to constraint-based programming, as such extending the work by Klotzbücher et al. [93]. MDE proposes a systematic approach to model a domain, using four M-levels of abstraction. This chapter follows the meaning given to the levels in [93]:

M3 Highest level of abstraction, model of the constraints that a valid CBP specification DSL should conform to.

M2 The level of the application-independent CBP specification DSL, as a parameterized template.

M1 The level of application-specific CBP specification DSL.

M0 The level of concrete implementations using software libraries and frameworks.

uMF [92], a declarative and light-weight metamodelling framework forms the M3 level, enabling the modelling and validation of structural constraints on the presented DSL. As for uMF, this chapter presents a Lua [74] based internal DSL, i.e. a DSL constructed on Lua as a host language. Lua is a light-weight scripting language, already integrated in several robotic software frameworks and DSLs, such as Orocos [33], ROS [171], and rFSM [94].

Figure 4.1 gives an overview of the M-levels and the software tools for constraint-based programming on each level. This chapter focuses on the constraint-based programming DSL named *itasc_dsl*, and gives an example model written in the DSL. The *itasc_dsl_orocos_deployer* is a software tool to instantiate a model written in the presented DSL using the iTaSC software framework.

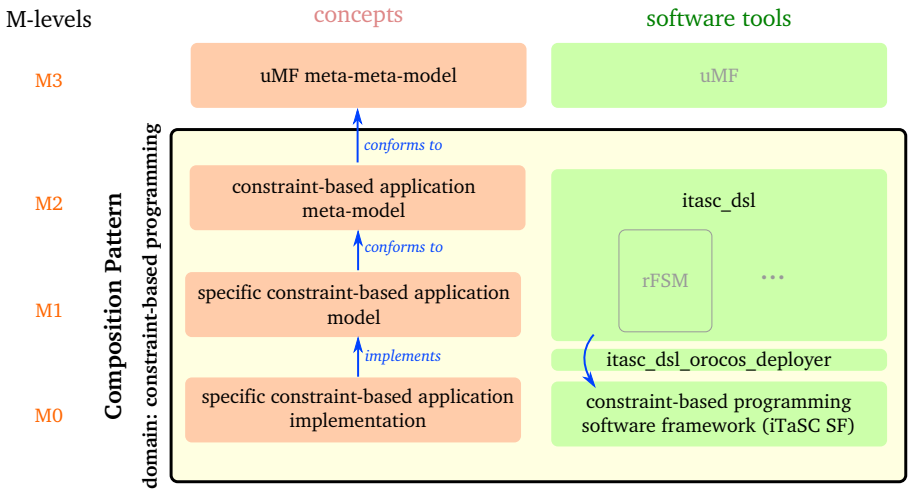


Figure 4.1: Overview of the M-levels and software tools for constraint-based programming on each level. The parts in grey indicate existing DSLs or software, such as uMF and rFSM.

The chapter first introduces the running example in Section 4.3, and then introduces the meta-model of the CBP specification DSL in Section 4.4. Next it elaborates on a model of an CBP specification in Section 4.5. Further it explains the transition from M1 to the executable code on M0 in Section 4.6. Section 4.7 discusses and evaluates the proposed DSL, and finally Section 4.9 summarizes the innovations and future work.

4.3 Running example

All concepts introduced in subsequent sections will be explained using the following example. The example consists of a drawer opening application with a PR2 robot as shown in Figure 4.2. The robot has to (i) reach for the handle with its right gripper, (ii) grasp the handle, and (iii) open the drawer, (iv) while keeping close to a preferable joint configuration, and (v) staying away from

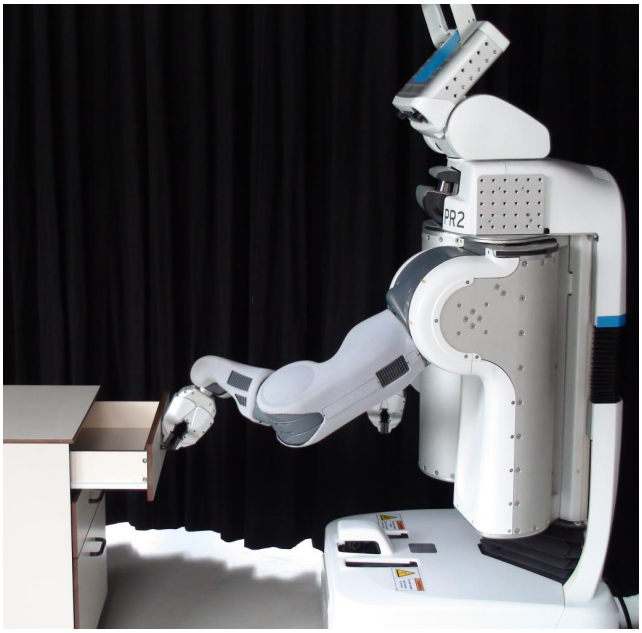


Figure 4.2: Setup of the drawer opening example.

joint limits. A video and the full model of the example can be found online at [164] and [163][†], respectively. Listings 4.1 - 4.4 show the model for the drawer opening part of the example, which will be explained in detail in the following sections.

4.4 Application-independent meta-model for constraint-based programming (M2)

The M2 model describes a *template* for a robotic constraint-based programming application. The systematic iTaSC workflow [46, 165, 166] eases the domain analysis, which identifies concepts and structures of the constraint-based programming domain.

The design workflow consists of six steps, and is briefly recapitulated here: (i) identify the *robots and objects* involved in the application and their location in the scene, (ii) define the *object frames* on the robots and objects at locations

[†]Appendix C.2 lists and details the videos related to this chapter.

where a task will take effect, (iii) parametrize the space between pairs of object frames, as a *virtual kinematic chain (VKC)* with the *feature coordinates* χ_f as joint coordinates, (iv) choose the *outputs* $\boldsymbol{y} = f(\boldsymbol{q}, \boldsymbol{\chi_f})$ to be constrained, (v) impose *constraints* on the relative motion between two object frames by selecting the type of constraints (equality or inequality) and the control law that enforces them, (vi) select a constraint-optimization problem solver that calculates the desired robot joint inputs. Figure 4.5 shows a schematic representation for the drawer opening task, including its kinematic loop. The kinematic loop consists of the VKC, the kinematic model of the robot, the object model (relation between named frames on the drawer) and scene model (placement of the robot and the drawer in the scene). Section 4.5 will explain the figure in more detail.

The iTaSC software framework reflects this systematic way of describing tasks. The implementation of the functionality follows the Composition Pattern described in Chapter 3. It builds upon the Orocos software component framework [33] and rFSM statecharts [89, 91, 94]. Furthermore, it integrates with ROS.

Building on the iTaSC theory and software concepts, we developed a first generation iTaSC DSL [167], integrating well established DSLs such as rFSM. This chapter discusses the second generation DSL, after redesigning the DSL following the Composition Pattern, which was introduced in Chapter 3. Section 4.8 will discuss how this redesign improved the DSL.

The structure of the DSL follows the composition tree of a constraint-based application shown in Figure 3.5. It gives an overview of the meta-model of a constraint-based application, which will be formalized in this chapter as a domain-specific language. However, the DSL does not formalize each branch of the tree shown in Figure 3.5 to the same depth level. Moreover, the Constraint-Output is left out, as will be discussed in Section 4.4.3. Figure 4.3 shows the composition tree that the DSL formalizes.

Furthermore, each entity represented in the DSL follows (conforms to) the Composition Pattern explained in Chapter 3 and shown in Figure 3.3. Therefore, the meta-model of a Composite Functional Entity refers to the meta-models of its child Functional Entities, and the support entities. The presented DSL will limit itself to only make the `Coordinator`, the `Configurator`, and (part of) the `Composer` of the support entities explicit, as will be discussed in Section 4.8.

- The `Coordinator` of a composite coordinates its behavior by communicating events, it is a pure event processor, independent of the other four *concerns*. The rFSM DSL [94] forms the meta-model of a `Coordinator`.

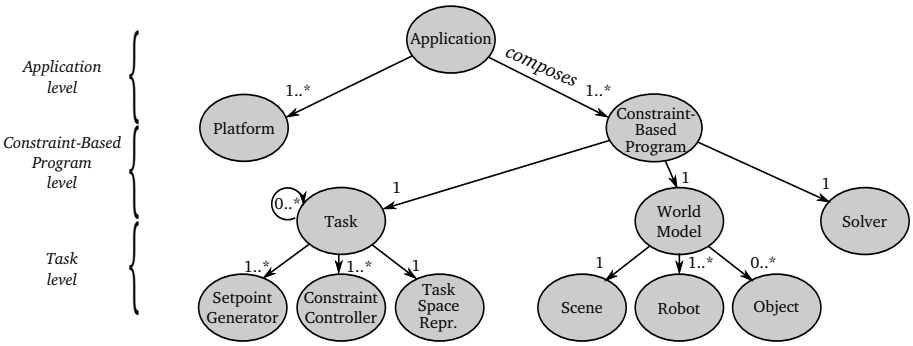


Figure 4.3: Composition tree of a robotic constraint-based application that is represented in the DSL. A node indicates an entity meta-model. An arrow indicates composition, pointing from the Composite Functional Entity to the Functional Entity that forms part of the composite. The numbers indicate the multiplicity, i.e. the range of possible number of models that are part of the composite and which conform to the designated meta-model. The different depth levels of the composition tree are named Application level, Constraint-Based Program level, and Task level, as indicated with the braces on the left.

- The `Configurator` applies settings to an entity of the composite it belongs to, when triggered by the `Coordinator`. The Configurator DSL [89] forms the meta-model of a `Configurator`. This meta-model is extended to allow for static configurations. These static configurations load the default configuration, and provide compatibility with the previous DSL version. The `Coordinator` and `Configurator` conform to the Configurator-Coordinator pattern of Klotzbücher et al. [89].
- The `Composer` specifies the interconnection of the entities within a composite. The `Composer` specifies 'configuration with structural consequences', where the `Configurator` specifies configuration without these structural consequences. The DSL is restricted to static compositions in specific variation points. The following sections will give detailed examples.

Next to the Functional Entities shown in Figure 3.5 and the Support Entities, the DSL adds following three attributes to each (Composite) Functional Entity:

- The `Name` identifies the entity within the model,
- the `uri` (Uniform Resource Identifier) uniquely identifies the model,
- and the `dsl_version` identifies the M2 model version.

Each entity can have two extra attributes

- a `type` attribute specifying the specific type of an entity, and
- the `package` attribute pointing to the software package where to find the implementation of this type.

These two attributes are mandatory for leaf entities.

Furthermore, certain composites (can) compose multiple entities that conform to the same meta-model. For example, an `Application` can compose multiple `Platforms`. Because of the Lua and uMF syntax, these entities are grouped in the DSL under an attribute with a lower-case name. For example, `platforms` groups different `Platforms`.

This chapter uses the same notation as in previous chapter: italic font names indicate *models*, and teletype font names indicate `meta-models`. For example a *pull_drawer_handle* `Task` denotes a *Task* model, conforming to the `Task` meta-model. Moreover, teletype font names are used to indicate a non-specific model that conforms to the meta-model with the same name, but only in places where this is clear from the context. For example a `Task` indicates a non-specific task model that conforms to the `Task` meta-model.

The structure of the DSL incorporates multiple *levels* of composition. To improve readability, this chapter names the upper three of these levels to one of its entities, from high to low level: `Application`, `Constraint-Based Program`, and `Task`, as shown in Figure 4.3. The following sections explain these three levels in detail.

To further increase readability, following subsections explain the meta-model (the DSL) by giving examples, rather than explaining its grammar. More concretely, these examples explain what a model that conforms to the meta-model specifies, or they explain the behavior of an implementation at runtime. For example, we state *'A `Platform` specifies the concrete resource to be used'*, to shorten *'A concrete model conforming to the `Platform` meta-model specifies the concrete resource to be used'* or *'A `Platform` meta-model provides a language to specify a concrete resource to be used'*.

4.4.1 Application level

The `Application` forms the highest level in the composition tree and composes entities that conform to following meta-models:

- A `Platform` specifies the concrete resource to be used. In practice, it includes the reference to the platform- and hardware interfaces, for example the interface to the PR2 controller manager of ROS.
- The `Constraint-Based Program` contains the model of the actual constraint-based task specification, for example all the constraints needed to reach, grasp, and open the drawer, and how these can be resolved. Next Section 4.4.2 explains the `Constraint-Based Program` in detail.
- A `Coordinator` coordinates the behavior of the full application for example the configuring and starting of the different platforms, etc.
- A `Configurator` provides settings for the different entities, for example a specific operation mode of a platform.
- A `Composer` connects the entities. Important is the relation of the `Platform` to the `Constraint-Based Program`. Therefore, following section will make this configuration explicit.

4.4.2 Constraint-based program level

The `Constraint-Based Program` composes the task to be executed with a world model and a solver. This task can be a composite of other tasks, together specifying a set of constraints on the world model. These constraints forms an over- and/or underconstrained [46] optimization problem which will be resolved by the solver. The `Constraint-Based Program` composes entities that conform to following meta-models:

- A `World Model` specifies the scene with the robots and objects involved in the application. The composition of a `World Model` will be explained in the next section.
- A `Task` specifies a set of constraints on the world model that form together an over- and/or underconstrained [46] optimization problem. It can be a composite of other tasks. The composition of a `Task` will be explained in the next section.
- A `Solver` contains the algorithm that solves the optimization problem for a certain objective function, taking the constraints of the the `Task` into account. This results in the desired inputs for the robot platforms, typically desired joint velocities, accelerations or torques. In the running example a prioritized, weighted damped-least squares solver [10, 69, 142] is used, solving for joint velocities. The objective is to minimize the error in task space and on the joint velocities.

- A `Coordinator` coordinates the interaction between the different entities, for example commanding a new operational mode for the solver.
- A `Configurator` configures the different entities, for example adapting the objective function for the solver to the task at hand, when commanded by the `Coordinator`.
- A `Composer` connects the entities. Important is the connection of the `Task` to the `World Model`. Therefore, following section will make this connections explicit.

4.4.3 Task level

A `World Model` composes entities that conform to following meta-models:

- A `Robot` models a robot involved in and controlled by the application, e.g. the PR2 in the drawer opening example. Each `Robot` provides a 'view' of a platform, e.g. the kinematic and dynamic structure of the robot. It forms the integration point for software or DSLs to represent kinematic or dynamic structures such as Collada [13] or URDF [172].
- An `Object` models an object involved in, but not controlled (actuated) by, the application, for example the cabinet with the drawer to be opened. An `Object` has the same structure as a `Robot`, but doesn't have controllable degrees-of-freedom (DOF).
- The `Scene` models the environment in which the robots and objects are located. It contains `SceneElements`, i.e. object frames in the scene, some of which can be configured.
- A `Coordinator` coordinates the interaction between the different entities, for example demanding a more accurate representation of an object.
- A `Configurator` configures the different entities, for example loading a more accurate representation of an object, when commanded by the `Coordinator`. Important is the configuration of the configurable `SceneElements` of the `Scene` (i.e. the `Locations` in the `Scene`). This `Location` can be fixed or an external input, provided by for example sensor information. Therefore, the following section will make this configuration explicit.
- A `Composer` connects the entities. Important is the connection of the `Robots` and `Objects` to the `SceneElements` in the `Scene`. Therefore, the following section will make this configuration explicit.

A (non-composite) `Task` composes entities that conform to following meta-models:

- The `VKC` models the task space as a *Virtual Kinematic Chain*, a specific form of a Task Space Representation with *feature coordinates* as joint coordinates. For example, the use case considers a cylindrical task space: TransZ, RotZ, TransX, RotX, RotY, RotZ[‡] for the *reach_drawer_handle* task, since the handle of the drawer is a cylinder and it is irrelevant from which side the robot approaches the handle. The use case considers a cartesian task space: TransX, TransY, TransZ, RotX, RotY, RotZ for the *pull_drawer_handle* task. The chain aspect forms also an integration point for software or DSLs to represent kinematic structures.
- The `CC` models the *Constraint-Controller* that imposes a desired value on an output, enforced by a controller. The output is a function of the controllable robot joints and feature coordinates. In the open drawer example, we use a simple proportional controller on the position error and velocity feedforward on each feature coordinate ($\boldsymbol{y} = \boldsymbol{\chi_f}$).
- A `Setpoint Generator` models a setpoint generator, which delivers desired values to the controllers in the application, for example a trajectory in task space to open the drawer. A `Setpoint Generator` can be very different in nature; as simple as a fixed value, complexer trajectory generators, or planners.
- The `Coordinator` coordinates the behavior of one task, for example enabling or disabling a single constraint of a task.
- The `Configurator` configures the different entities, for example the control gains of the `CC`.

Moreover, a `Task` can be a composite of other `Tasks`, in this case the `Configurator` and `Coordinator` have extra responsibilities:

- The `Coordinator` coordinates the composite task behavior by enabling and disabling tasks, changing weights and priorities, etc. For example disabling the reaching task and activating the grasping task once an event is received that signals that the handle is reached.
- The `Configurator` configures a `TaskSetting` for each `Task`. The `TaskSetting` assigns a `weight` and a `priority` to the `Task`, two

[‡]Trans means translation and Rot rotation, along the direction or around an axis of the moved coordinate frame.

measures to deal with over- and/or under-constrainedness of the composite task [110,142]. In the running example the task to stay close to a preferable joint configuration has a lower priority than the reaching motion.

4.4.4 Decoupling

§ The presented DSL decouples the constraint-based programming application in many ways. For example, a `Task` model is independent from (i) the `weight` or `priority` that is assigned to the `Task`, or (ii) the object frames in between which a `Task` is assigned.

Note the separation of the Configuration in `Configurator`, the Coordination in `Coordinator`, the Composition in the `Composer` and the functionality (Computation) in the different `Functional Entities`.

4.5 An iTaSC model (M1)

The M1 level model *conforms to* the M2 meta-model, filled in with the application-specific information. Due to the limited space, we restrict the example code to the model for the composite task of the drawer opening part of the use case, as listed in following sections. The full model can be found on [163].

The M2 model and uMF [92] tools enable formal verification of the conformity of the M1 model to the M2 model. This verification comprises syntax verification, the existence of referred entities¶ and DSLs, and compatibility between entities. The automatic verification returns meaningful errors to the user, as will be shown in Section 4.7.

A model of a Composite Functional Entity is indicated in the DSL by the name (or abbreviation) of its meta-model, with the attributes and entities it composes between braces (a Lua table). As explained in Section 4.4, a composite has a `Name`, `uri`, and `dsl_version` attribute, and optionally a `uri` and `dsl_version` attribute. The attributes and entities have different Lua table keys and corresponding value (`key = value`). The keys of a `Coordinator`, `Configurator`, and `Composer` are `coord`, `conf`, and `comp`, respectively. Values starting with 'my_', indicate a Composite Functional Entity (or group of Composite Functional Entities) that is detailed further on in the text.

§A more elaborate discussion on decoupling by applying the Composition Pattern is given in Chapter 3.

¶`Composers` refer to entities by their `Name` attribute.

4.5.1 Application level

Listing 4.1: Application level

```
return Application {
name = "simple_open_drawer_using_real_pr2",
dsl_version = "0.2",
uri = "be.kuleuven.mech.rob.app.drawer_simpr2",
platforms = { Platform{
  name = "pr2_hardware",
  dsl_version = "0.2",
  uri = "be.kuleuven.mech.rob.platform.pr2",
  file = "file://pr2driver.lua"
  }},
cbp = my_constraint_based_program,
coord = Coordinator{fsm = "file://app_supervisor.lua"},
config = Configurator{conf = "file://↩
    simple_open_drawer_using_real_pr2.cpf"},
compo = Composer{ entity_conns = {
  EntityConn{
    name = "conn_pr2hw_pr2",
    e1 = "pr2_hardware",
    e2 = "itasc_simple_open_drawer_pr2.experimental_setup.pr2"}}}}
```

The *simple_open_drawer_using_real_pr2* `Application` composes two `Functional Entities`: a *pr2_hardware* `Platform` and a `Constraint-Based Program` which will be detailed in the next section.

The *pr2_hardware* `Platform` in the presented example is a leaf entity, hence it has a `package` and `type` attribute. All referred `packages` and `types` can be found in [163].

The `Coordinator` of the presented `Application` is an entity that loads an rFSM [89, 91, 94] finite-state machine description. The `Coordinator` of all composites share the same underlying structure, as shown in Figure 4.4. Each of the states can be a state machine on its own, as will be detailed in Chapter 5.

The `Configurator` of the presented `Application` has (static) configuration, contained in an xml file, with a configuration property file (cpf) extension.

The `Composer` models an entity connection, in this case connecting a specific *pr2_hardware* `Platform` to the *pr2* `Robot`.

4.5.2 Constraint-based program (CBP) level

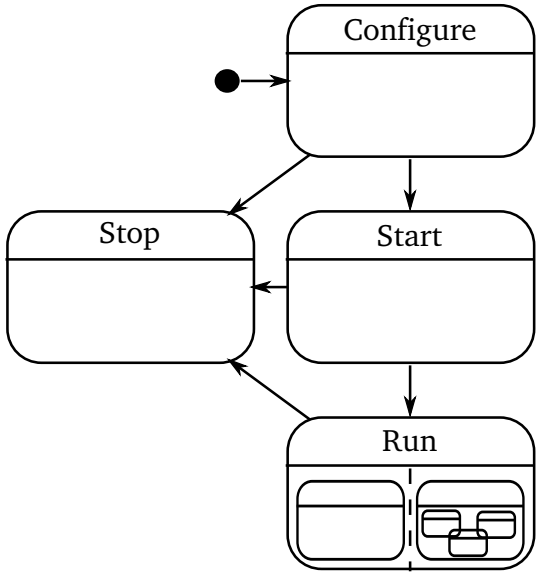


Figure 4.4: Basic infrastructure of a `Coordinator` of a level. Each state is possibly a (combination) of state machines, as shown for the Run state.

Listing 4.2: CBP level

```
my_constraint_based_program = CBP {
name = "itasc_simple_open_drawer_pr2",
dsl_version = "0.2",
uri = "be.kuleuven.mech.rob.cbp.drawer_pr2",
world_model = my_world,
solver = Solver{
  name = "wdls_prior_vel_solver",
  dsl_version = "0.2",
  uri = "be.kuleuven.mech.rob.solver.wdls_prior_vel_solver",
  package = "wdls_prior_vel_solver",
  type = "iTaSC::WDLSPriorVelSolver"},
task = my_task,
coord = Coordinator{fsm = "file://cbp_supervisor.lua"},
config= Configurator{conf = "file://cbp_supervisor.cpf"},
compo = Composer{ object_frame_conns = {
  ObjectFrameConn{
    name = "conn_pull_cabinet",
    fr1 = "simple_open_drawer.pull_drawer_handle.o1",
    fr2 = "experimental_setup.cabinet.upper_drawer"},
  ObjectFrameConn{
    name = "conn_pull_rgripper",
    fr1 = "simple_open_drawer.pull_drawer_handle.o2",
    fr2 = "experimental_setup.pr2.r_gripper_tool_frame"},
```

```
  ObjectFrameConn{
25  name = "conn_keepjnt_pr2",
    chain = "simple_open_drawer.keep_joint_config",
    robot = "experimental_setup.pr2"}}}}
```

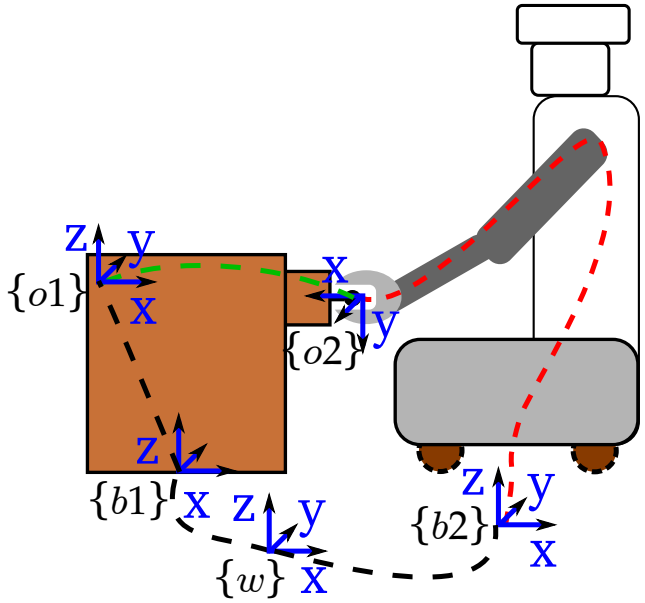


Figure 4.5: The different parts of the kinematic loop (dashed lines) for the *pull_drawer_handle* `Task`. The green line indicates the Virtual Kinematic Chain on which the task constraints are imposed, black lines indicate fixed kinematic relations, and red lines indicate the controlled robot joints. The *robolab* `Scene` is attached to the world $\{w\}$. Two `SceneElements` (frames) are defined in the *robolab*: *start_loc* and *cabinet_pos*. The *cabinet* `Object` *base* frame $\{b1\}$ is attached to –hence coincides with– the *cabinet_pos* frame. The *pr2* `Robot` *odom* frame $\{b2\}$ is attached to the *start_loc* frame. The Virtual Kinematic Chain is attached to two object frames $o1$ and `o2`. The first object frame $o1$ is defined as (coincides with) the *upper_drawer* of the *cabinet*, the second object frame `o2` is defined as the *r_gripper_tool_frame* of the *pr2*. The *upper_drawer* object frame ($o1$) locates where the drawer fits in the cabinet, it is fixed to the cabinet and does not move with the drawer itself. The *r_gripper_tool_frame* object frame (`o2`) coincides with the *handle* of the *cabinet*, since it is previously grasped by the robot.

The *itasc_simple_open_drawer_pr2* `Constraint-Based Program` composes three `Functional Entities`: a *World Model*, a `Solver`, and a

Task. The next section details the *World Model* and the Task. The *wdls_prior_vel_solver* Solver is a prioritized, weighted damped-least squares solver.

The Composer explicitly models the connection of the Task to the World Model. More concretely, these models are connected by attaching object frames defined on the World Model with object frames defined on the Task, as shown in Figure 4.5. For example, the *pull_drawer_handle* sub-Task of the *simple_open_drawer* composite Task is connected to the upper drawer of the *cabinet* Object and the right gripper of the *PR2* Robot respectively. The *pull_drawer_handle* Task doesn't have a connection to a Robot, since there are no constraints in joint space. On the other hand, the *keep_joint_config* sub-Task of the *simple_open_drawer* composite Task is connected to the whole Robot since it constrains the Robot's joint space.

4.5.3 Task level

Listing 4.3: Task level: World Model

```
my_world = WorldModel {
name = "experimental_setup",
dsl_version = "0.2",
uri = "be.kuleuven.mech.rob.worlds.experimental_setup",
robots = { Robot{
  name = "pr2",
  dsl_version = "0.2",
  uri = "be.kuleuven.mech.rob.robots.pr2"
  package = "itasc_pr2",
  type = "iTaSC::pr2Robot"
}},
objects = { Object{
  name = "cabinet",
  dsl_version = "0.2",
  uri = "be.kuleuven.mech.rob.objects.furniture.cabinet"
  package = "fixed_object",
  type = "iTaSC::FixedObject"}},
scene = Scene{
  name = "robolab",
  dsl_version = "0.2",
  uri = "be.kuleuven.mech.rob.scene.robolab",
  package = "scenes",
  type = "iTaSC::Scene"
},
coord = Coordinator {fsm = "file://experimental_setup.lua"},
config = Configurator{ scene_elements = {
  SceneElement {
    name = "start_loc",
    location = Frame {
```

```
        M = Rotation{X_x=1,Y_x=0,Z_x=0,X_y=0,Y_y=1,Z_y=0,X_z=0,Y_z=0,Z_z=1},
        p = Vector{X=0.0,Y=0.0,Z=0.0}}},
  SceneElement {
    name = "cabinet_pos",
    location = Frame {
      M = Rotation{X_x=1,Y_x=0,Z_x=0,X_y=0,Y_y=1,Z_y=0,X_z=0,Y_z=0,Z_z=1},
      p = Vector{X=2.8,Y=0.0,Z=0.5}}}},
  conf = "file://experimental_setup.cpf"},
compo = Composer{ object_frame_conns = {
  ObjectFrameConn{
    name = "conn_w_pr2odom",
    fr1 = "robolab.start_loc",
    fr2 = "pr2.odom"},
  ObjectFrameConn{
    name = "conn_w_cabinetbase",
    fr1 = "robolab.cabinet_pos",
    fr2 = "cabinet.base"}}}}
```

The *experimental setup* `World Model` composes the *PR2* robot and the *cabinet* with the drawer to open.

For the drawer opening example, the object frames in the *robolab* `Scene`, i.e. the `SceneElements`, are configurable `Locations`. A `Location` is expressed in the uMF frame specification, consisting of a rotation matrix and position vector. For example, the *start_loc* `SceneElement` is connected to the *odom* frame on the *PR2* robot, as shown on Figure 4.5. It defines the start location of the *PR2* robot. Remark that in the current implementation the *odom* frame will stay fixed in the *start_loc* location. It forms the starting point of the kinematic chain describing the *PR2* robot, since the location of the *PR2* robot in the *robolab* `Scene` is part of the 'controllable DOF'.

The `Composer` explicitly models the connection of the `Robots` and `Objects` to the `Scene`. More concretely, these models are connected by attaching object frames defined in the `Scene` with object frames defined on the `Robots` and `Objects`. These connections have the same meta-model as the connection of the `Task` to the `World Model`.

Listing 4.4: Task level: (Composite) Task

```
my_task = Task{
  name = "simple_open_drawer",
  dsl_version = "0.2",
  uri = "be.kuleuven.mech.rob.task.simple_open_drawer",
  tasks = {
    my_pull_drawer_handle,
    my_keep_joint_config},
  coord = Coordinator{fsm = "file://composite_task_supervisor.lua"},
```

```
  config = Configurator{
    conf = "file://simple_open_drawer.lua",
    task_settings = {
      TaskSetting{
        task = "pull_drawer_handle",
        weight = 1.0,
        priority = 1},
      TaskSetting{
        task = "keep_joint_config",
        weight = 1.0,
        priority = 2}}}}
```

The composite *simple_open_drawer* `Task` composes two `Tasks`:
(i) *pull_drawer_handle*, to pull open the drawer and (ii) *keep_joint_config*, to keep a preferred joint configuration. These `Tasks` are detailed further in this section.

The `Coordinator` of the *simple_open_drawer* `Task` is rather limited for the running example, since all tasks are running in parallel during this single opening action. The full drawer opening application needs more complicated, multi-state coordination at run-time [163].

The `Configurator` of the *simple_open_drawer* `Task` assigns weights and priorities to each sub-`Task`. `Tasks` with a lower priority number have priority over `Tasks` with a higher priority number. In the running example, the *pull_drawer_handle* `Task` has priority over the *keep_joint_config* `Task`. The weights will have no effect in this reduced example, since there are no conflicting constraints within each priority level.

Listing 4.5: pull_drawer_handle Task

```
my_pull_drawer_handle = Task{
    name = "pull_drawer_handle",
    dsl_version = "0.2",
    uri = "be.kuleuven.mech.rob.task.pull_handle",
    vkc = VKC{
      name = "cartesian_coordinates",
      dsl_version = "0.2",
      uri = "be.kuleuven.mech.rob.vkc.cartesian",
      type= "iTaSC::VKC_sixDof",
      package = "sixDof_pff",
      chain = {"TransX","TransY","TransZ","RotX","RotY","RotZ"}},
    cc = CC{
      name = "sixdof_pff",
      dsl_version = "0.2",
      uri = "be.kuleuven.mech.rob.cc.sixdof_pff",
      type="iTaSC::CC_sixDof_pff",
      package = "sixDof_pff"},
    sg = SetpointGenerator{
      name = "open_drawer_trajectory_generator",
```

```
      dsl_version = "0.2",
      uri = "be.kuleuven.mech.rob.sg.open_drawer",
      type = "trajectory_generators::nAxesGeneratorPos"
      package = "naxes_joint_generator"},
    coord = Coordinator{
      fsm = "file://sixDof_pff_supervisor.lua"},
    config = Configurator{
      conf = "file://pull_drawer_handle.lua"}}
```

The *pull_drawer_handle* `Task` composes three `Functional Entities`: a `CC`, `VKC`, and a `SG`. The `CC`, `VKC`, and `SG` are leaf entities, specifying a `package` and a `type` which can be found in [166]. The *cartesian_coordinates* `VKC` models a Virtual Kinematic Chain with a cartesian coordinate system. The *sixDof_pff* `CC` refers to a simple proportional controller with feed-forward for six DOF output $\boldsymbol{y}$, in this case the feature coordinates of the *cartesian_coordinates* `VKC`. The *open_drawer_trajectory_generator* `SG` generates a trajectory to open the drawer.

Listing 4.6: keep_joint_config Task

```
my_keep_joint_config = Task{
    name = "keep_joint_config",
    dsl_version = "0.2",
    uri = "be.kuleuven.mech.rob.task.keep_jnt_cfg"
    cc = CC{
      name = "pdff_joints",
      dsl_version = "0.2",
      uri = "be.kuleuven.mech.rob.cc.pdff_joints",
      type = "iTaSC::CC_PDFFjoints",
      package = "joint_motion"},
    sg = SetpointGenerator{
      name = "desired_joint_config_generator",
      dsl_version = "0.2",
      uri = "be.kuleuven.mech.rob.sg.joint_config",
      type = "trajectory_generators::nAxesGeneratorPos"
      package = "naxes_joint_generator"},
    coord = Coordinator{
      fsm = "file://jnt_config_supervisor.lua"},
    config = Configurator{conf = "file://keep_joint_config.cpf"}}
```

The *keep_joint_config* `Task` composes two `Functional Entities`: a `CC` and a `SG`. The `CC` and `SG` are leaf entities, specifying a `package` and a `type` which can be found in [166]. The *keep_joint_config* `Task` has no `VKC`, since all constraints are in joint space. The *CC_PDFFjoints* `CC` refers to a general PD controller with feed-forward for an n-DOF output $\boldsymbol{y}$. The *desired_joint_config_generator* `SG` generates a trajectory for each joint towards

a (fixed) desired position.

4.6 Code generation: from M1 to M0

The M1 model *specifies* a robot application, that has to be transformed into an *implementation* that conforms to this M1 model. The model to meta-model conformity can be checked using the software tooling for uMF [92]. We provide software support that transforms the M1 model to a run-time configuration and instantiation using the existing iTaSC software implementation [166]. This iTaSC software is developed using the Orocos component framework. Chapter 5 details the implementation aspects of the iTaSC software framework.

Figure 4.1 gives an overview of the software tools. The implementation of the DSL can be found online at `http://bitbucket.org/dvanthienen/itasc_dsl`. The *itasc_dsl_orocos_deployer* software tool to instantiate a model into an iTaSC software framework implementation can be found online at `http://bitbucket.org/dvanthienen/itasc_dsl_orocos_deployer.git`.

4.7 Experiments and evaluation

Table 4.1 compares the required lines of code for two more elaborate examples from previous work: lissajous-tracing with a KUKA youBot [162] and human-robot comanipulation with a PR2 robot [165], detailed in Section 7. The table shows the lines of code (LOC) to be hand coded to generate the application using the DSL (*model*) or the iTaSC software framework (*original code*). Since a model written in the DSL is instantiated using the iTaSC software framework, both implementations share the iTaSC Orocos components needed for the execution (not included in the LOC). Next to this code, both share the rFSM models, and the component configuration values, which are not counted in the LOC.

When using the DSL, i.e. when modeling the application, the developer can start from a model template (included in the LOC), which he (or she) has to complete with the application specifics. When using the iTaSC software framework directly, the developer has to provide scripts to create the application, including scripts that provide application specific component instantiation, connection, extra‖ configuration, and extra coordination.

‖'Extra' with respect to the configuration and coordination shared with the DSL version.

The total lines to be hand coded have reduced by a factor of 2.5 when using the DSL. This reduction is possible by the automatic derivation of framework specific code from the model. Moreover the model provides a better readable overview of the application and introduces names in a more consistent 'hierarchical' manner: lower levels introduce names, referenced to by higher levels. In order to allow this referencing, each level has to expose the names of the entities it composes to the higher levels.

	laser tracing	comanipulation
model	97	155
original code	237	416

Table 4.1: Comparison of code efficiency by lines of code of a laser tracing and comanipulation example.

The warnings and errors that the execution of the M1 model verification returns include:

- Syntax errors, such as the misspelling of an attribute or entity, or the assignment of a wrong type. For example when erroneously using `ame = 'pull_drawer_handle'` in stead of `name = 'pull_drawer_handle'` when assigning a name to the first task:

```
err@ app.cbp.tasks[1].ame:
  illegal field 'ame' in sealed dict
  (value: pull_drawer_handle)
err@ app.cbp.tasks[1]:
  non-optional field 'name' missing
```

  or when assigning a number to the `name`:

```
err@ app.cbp.tasks[2].name:
  not a string but a number
```

- The non-existence of referred entities and DSLs, for example robots not listed in `Robots` or not found configuration files:

```
err@ app.cbp.worldmodel: failed to resolve ObjectFrameConn.fr2↩
    :
  pr3.odom
err@ app.config:
  non-existing configuration file ↩
      simple_open_drawer_using_real_pr2.cpf
```

- Incompatibility between entities, for example when assigning an always singular Virtual Kinematic Chain with two Z rotational joints as first two joints:

```
1 err@ app.cbp.task.tasks[1].vkc.chain:
    identical consecutive chain segments (1-2)
```

- The use of the same name for multiple entities, for example when giving the object and the robot the name 'pr2':

```
1 err@ : duplicate use of name pr2
```

- The use of an outdated version of the meta-model:

```
1 warn@ app.cbp.dsl_version:
  Current CBP meta-model version number 0.1,
  does not match required version number 0.2
```

One of the major advantages of the developed DSL is its ease to create and adapt applications. As a proof, the following paragraphs summarize some possible changes of the running example. Video fragments of some of the changes can be found at [164].

To **change the robot** that is used to a totally different platform, as in the case given in Section 4.2, one has to change:

- the `Robot` (Listing 4.3, line 6,8-10) to for example the KUKA YouBot,
- the robot `Platform` (Listing 4.1, line 6-9),
- the connections that use object frames defined on this robot,
  - *conn_pr2hw_pr2* (Listing 4.1, line 16-18),
  - *conn_pull_rgripper* (Listing 4.2, line 23),
  - *conn_keepjnt_pr2* (Listing 4.2, line 27),
  - *conn_w_pr2odom* (Listing 4.3, line 42),
- the configuration of joint space tasks, such as the *keep_joint_config* `Task` (Listing 4.6, line 19).

A total of 15 minor modifications are needed to change the robot platform used, far less than the more than 100 lines without the model. The same reasoning holds for changing an `Object`. There will be even fewer changes needed, since there is no need to change a `Platform`.

Another common **change** to an application is the **relation between the tasks** by changing the `weight` and `priority` of one or multiple tasks. These settings are grouped in the `Configurator` of the *simple_open_drawer* composite (Listing 4.4, line 12-19). It is common that these settings are commanded to change at run-time by the `Coordinator` of the composite.

In case the drawer does not slide but swivels open like a door, one can **adapt the model of the task space** easily, by changing the `chain` of the `VKC` (Listing 4.5, line 11) to cylindrical coordinates. Cylindrical coordinates ease the task specification on a swiveling door to a constraint on a single DOF, namely the angle around its pivot.

Another common alteration is the **change of controller** used for a task, which is easily done by replacing the `CC` of the task. For example, replacing the proportional controller of the *pull_drawer_handle* `Task` to an impedance controller (Listing 4.5 lines 13-17).

In case one wants to **change the coordination of the composite task**, one only has to change the `Coordinator` of the *simple_open_drawer* (composite) `Task` (Listing 4.4, line 8). For example deactivating the *keep_joint_configuration* `Task` once the handle is grasped, or change how and when the transition from grasping the handle to opening the drawer occurs.

In case the lower drawer, rather than the upper drawer must be opened, one can easily **change the object frame** to this other drawer by changing one word (Listing 4.2, line 19). Further, one can easily change which of the two arms of the robot should be used, by simply changing one word, namely the object frame on the robot; for the running example replace `r_gripper_tool_frame` by `l_gripper_tool_frame` (Listing 4.2, line 23).

4.8 Discussion

This section first discusses how the application of the Composition Pattern to refactor the first DSL version [167] improved this DSL. Secondly, it discusses the relation with the TFF-DSL of Klotzbücher et al. [93].

4.8.1 Refactoring of the first (iTaSC) DSL

The first DSL did separate the application in a hierarchy of different entities, and it did separate the concerns following the 5Cs principle [32] to a large extent. However, it did not follow the Composition Pattern:

- It confused configuration with composition. For example, task weights and the connection of the object frames of a task to the world model were both attributes of a separate entity. This confusion resulted in many references throughout the design.
- The configuration of an entity was held in the entities itself. However, configuration depends on the relation of an entity to the other entities in a composite, hence this configuration should be set by the `Configurator` of the parent of the entity. For example, a specific gain of a `Constraint-Controller` (`CC`) depends on the task, hence is set by the `Configurator` of the `Task`.
- Each composite had a different structure in the first DSL version: some had (multiple) extra entities for configuration, some not; some had a `Coordinator`, some not; etc. The Composition Pattern made the structure uniform, following the layout shown in Figure 3.3, and it made the 'responsibilities' of the different entities clear (for example strict conformance to the configurator-coordinator pattern discussed in Chapter 3). As a consequence the overall hierarchy is the same as shown in Figure 3.5.

The presented DSL focused on the composition tree of constraint-based programming and the recursive application of the Composition Pattern. However, the presented DSL does not incorporate all aspects of the Composition Pattern (yet). It made only the `Coordinator`, `Configurator`, and (part of) `Composer` of the support entities explicit. The `Composer` is currently (largly) static, and does not model all interconnections. Moreover, the `Scheduler` and `Communicators` are not made explicit. The iTaSC software framework provides a boiler plate scheduler for each type of entity. Also the `Monitor` is not included in the DSL, the iTaSC software framework provides plug-ins for monitors, which are however not modeled. Future work should create general DSLs for these entities, and integrate them with the presented DSL.

4.8.2 From TFF DSL to CBP DSL

The work by Klotzbücher et al. [93] defines a DSL for the task-frame formalism, a special case of constraint-based programming using a single task frame (typically the tool centre point (TCP) frame or base frame) to define hybrid force position/velocity control operations. The authors distinguish a task-frame formalism model, a coordinator model, and the platform related configuration. These map, respectively, to the the composite `Task` model, `Coordinator`, and `Configurator` of the composite `Task` in the DSL presented in this chapter.

All other entities of an application are unmodeled or hidden, for example the composite `Task` model assumes different `Tasks`, each with following child entities:

- the `CC` is a hybrid force-position/velocity controller,
- the `VKC` is a Cartesian task space representation,
- and the `Setpoint Generator` is a fixed force, position, or velocity setpoint generator.

This TFF DSL model can however be transformed to a `Task` model within the here presented DSL, i.e. a model-to-model transformation. As a result, (i) all the task frame formalism related assumptions, and (ii) all the other entities of an application, are made explicit.

4.9 Conclusions and future work

This chapter structures and formally models constraint-based programming tasks using a domain-specific language (DSL). The presented DSL makes application development easier, yet is powerful to execute. Furthermore, the DSL enables automatic model verification, i.e. meta-model conformance, and code generation for different hard- and software platforms, diminishing code debugging efforts. However, the current implementation provides only limited model verification. More iterations are needed to provide DSL for the whole composition tree shown in Figure 3.5 and additional, lower or higher depth levels in this tree. Moreover, more relations between the `Functional Entities` can be (meta)-modeled and hence verified. In addition, the code generation is limited to the only supported software framework, the iTaSC software framework. Future work could provide code generation for more hard- and software platforms.

Furthermore, it is shown that the needed code and hence development time of constraint-based programming applications can be significantly reduced. Next to reduced code size, the rapid development originates from (i) the DSL as a scripting language, without need for compilation, (ii) the separation of concerns, leading to a structured set of small configuration files that are easily adapted, (iii) and the DSL as a template, guiding the programming effort.

The DSL is (re)developed using the Composition Pattern described in Chapter 3. Therefore, it separates concerns and groups reusable functionality, allowing non-experts to develop applications by *composing* tasks and assigning them to robots and objects in the scene. As such, it can be viewed as a first step towards the

robot programming language of the future. Moreover, the structured approach and DSL allows to integrate iTaSC in graphical programming tools, such as ABB's RobotStudio [1].

The proposed model opens up the possibility of tool support for design time model checking, using for example Xtext [56]. Further it allows the creation of a repository or store with models and/or implementations of entities, such as tasks i.e. a 'task store'.

Future work will focus on the integration of dedicated DSLs for all entities. Furthermore, the instantiation of the model on other constraint-based programming frameworks such as Stack of Tasks [102] will be investigated. Additionally, the presented formal modelling of constraint-based programming paves the way for robots to generate their own behavior, and reason on their behavior on a symbolic level. However, the current implementation does only help human developers to create applications faster. More work is needed to (meta)-model more concepts in robotics in order to enable robots to generate their own behavior.

The software implementation of the DSL and a software tool to instantiate a model using the iTaSC software framework is made available under an open-source license.

Chapter 5

The 5C-based Architectural Composition Pattern: Lessons Learned from re-developing the iTaSC framework for Constraint-Based Robot Programming *

5.1 Abstract

The authors are part of a research group that had the opportunity (i) to develop a large software framework (±five person year effort), (ii) to use that framework ("*iTaSC*") on several dozen research applications in the context of the specification and execution of a wide spectrum of mobile manipulator tasks, (iii) to analyse not only the functionality and the performance of the software but also its readiness for reuse, composition and model-driven code generation, and, finally, (iv) to spend another five person years on re-design and refactoring.

*Vanthienen, D., Klotzbuecher, M., Bruyninckx, H. (2014), "The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming", *Journal of Software Engineering for Robotics (JOSER)*, 5 (1), 17-35.

This chapter presents our major *lessons learned*, in the form of two best practices that we identified, and are since then bringing into practice in any new software development: (i) the *5C meta model* to realise *separation of concerns* (the concerns being Communication, Computation, Coordination, Configuration, and Composition), and (ii) the *Composition Pattern* as an architectural meta model supporting the methodological coupling of components developed along the lines of the 5Cs.

These generic results are illustrated, grounded, and motivated by what we learned from the huge efforts to refactor the *iTaSC* software, and are now behind all our other software development efforts. In the concrete *iTaSC* case, the Composition Pattern is applied at three levels of (modeling) hierarchy: application, iTaSC, and task level, each of which consist itself of several components structured in conformance with the pattern.

5.2 Introduction

Robotics has evolved from a single manipulator arm to a broad field of fixed, driving, crawling, diving, sailing and flying robots with many, redundant degrees-of-freedom (DOF). Each of them equipped with a wide range of sensors, from simple encoders to point cloud generating laser scanners. Moreover, more and more different, concurrently active tasks are integrated on these platforms in ever more demanding scenarios, such as human-robot co-manipulation.

One of our research priorities is the development of a systematic approach to program such complex tasks—i.c. the *instantaneous Task Specification and estimation using Constraints* (iTaSC) [46]—and to provide developers with appropriate software support to facilitate reuse [166]. This chapter focuses on what we learned along the way, as "best practices", to realise such large-scale software frameworks; these insights have been re-applied to the iTaSC software support context, which we use as a concrete application domain in this document, to make the generic, application-independent "best practices" more tangible, and the discussion about its pros and cons more concrete.

The focus of this chapter is not on discussing the *functionalities* offered by the iTaSC framework or any of the frameworks mentioned in the related work section; nor on discussing their relative merits, but on their software engineering design. The outcome is a set of "best practices" on how to tackle future labour intensive software development efforts, such that they could be developed with less pain, and integrated better with other frameworks.

One of the *major lessons learned* by the authors, is that integration should

not start at the level of the software code, but at the level of *models* of the provided functionality. In other words, the essential role of formal *domain-specific languages* (DSLs) will be stressed and illustrated at several occasions in the document. Remark that a transformation between models, and the generation of code from a model does not imply a "one to one" mapping; it can include optimizations based on "reasoning" on the model. A well known example of this principle are software compiler optimizations. In general such "model-to-x" transformations are a far from resolved problem, beyond the scope of this chapter.

While this work refrains from introducing "the" best system architectures, it does propose an *architectural pattern* (or "meta architecture") that has proven to be a "best practice" to help developers in finding and expressing the (most often rather complex) system architecture that fits best to their application's particularities.

5.2.1 The 5Cs

The software pattern introduced in this chapter builds on the *5Cs principle of separation of concerns* [32, 128] separating the **c**ommunication, **c**omputation, **c**oordination, **c**onfiguration, and **c**omposition aspects in the overall software functionality. This earlier work reflects our insights, or "analysis" of the design problem, while this chapter introduces our solution, or "synthesis", of how to provide constructive guidelines to system and component developers.

The authors consider the 5Cs as their most often proven "best practice" in robotics software development, since it (gradually) emerged during the huge accumulated software development experience (Section 5.3.4), and was applied to dozens and dozens of new software developments. Since two years, it is even the core of a course on *Embedded Control Systems* for first-year Master students in Mechanical Engineering, where it has proven essential to let them grasp, quickly and thoroughly, the high-level design challenges of a complex *system-of-systems*.

5.2.2 Outline and notation

Section 5.3 cites the related work and introduces the application domain. Sections 5.4–5.8 elaborate each of the five "5C" concerns, with a sub-section devoted to *modeling*, one on the *implementation*, one on discussion and lessons learned, and one on how to compose that concern in a bigger architecture. Section 5.9 states the conclusions of this chapter.

The chapter emphasizes entity[†] type names using `teletype font`, and instance names with *italic font*; names of events are emphasized using `teletype font` and begin with `e_`.

5.3 Related Work

This section gives an overview of related work and introduces the application domain. It further states the experience that led to the formulation of the Composition Pattern.

5.3.1 Robot Systems Architectures and Frameworks

Different *architectures* and *frameworks* have been proposed to create large and complex robot systems, an overview can be found in the book chapter by Kortenkamp and Simmons [95]. This section discusses some relevant and more recent work.

A first set of frameworks use hierarchical (concurrent) flow charts or state machines to create large and complex robot systems [119]. Recent examples include *ROSCo* [116] and *LightRocks* [156]. The latter focuses on task specification and will be discussed in Section 5.3.2.

Many robotic frameworks start from a multi-tiered architecture [22]. A recent two-tiered architecture, *robAPI* [8] aims at industrial robot applications. The first tier provides a real-time dataflow, and the second tier provides an object-oriented robotics API making abstraction of the real-time aspects, and dividing an application in actuators, actions, sensors, and state. Another example is the *BIP* (behavior, Interaction, Priority) framework [2,19], which has a three-tiered architecture. It provides *formal models* for the discrete behavior, which allows for *Validation and Verification* of those parts of the robot task.

Recently, cognition-enabled approaches have gained more attention. For example *CRAM* [16], a light-weight reasoning mechanism that can infer control decisions. It is a two-tiered architecture, merging the planning and sequencing layers of 3T architectures [22]. Another example of a cognition-enabled approach is the formal framework and agent-based software architecture by Doherty et al. [54].

[†]Entities, or components, agents, objects, modules, processes, activities... The concrete name has no real importance in the context of this chapter.

5.3.2 Application Domain: Task Specification

This subsection introduces the basic primitives of the application domain—specification and execution of complex robot tasks—that was chosen in this chapter to illustrate the best practices in software development for large-scale robotics software frameworks. This introduction is not meant to be self-contained or exhaustive, hence the reader is referred to the references for further details.

Traditionally, robot programming methods specify the robot motion in either joint space or Cartesian space. In joint space the motion trajectory is directly imposed on the individual robot joints, and is often used for programming fast point-to-point motions. In Cartesian space, for example used for tool trajectory tracking, the robot motion is specified in a *compliance frame* [104], or *task frame* [31] (typically either a tool centre point (TCP) frame or a base frame). Besides motion-based control, also joint-specific, Cartesian wrench (i.e. force and torque), and impedance control schemes are often used in practice [141].

This approach has proven its effectiveness for (geometrically) simple tasks, however, it scales poorly to more complex tasks that involve multiple frames and multiple partial motion specifications [31].

Constraint-based programming on the other hand does not consider the robot joints nor the single task frame as the central primitives in the specification. Instead, the core idea is to describe a *robot task as a set of constraints* (in *various* frames on the robot, in joint space as wel as in Cartesian or sensor space), and *one or more objective functions to optimize*. Samson et al. [137] presents this approach in a generic way, and De Schutter et al. [46] were the first to turn these generic ideas into a publicly available software framework. The latter, named **i**nstantaneous **Ta**sk **S**pecification using **C**onstraints (iTaSC), introduces particular sets of auxiliary coordinates to model uncertainty and to express task constraints. These task constraints are defined between *object frames* defined on robots and objects involved in an application. These object frames have, preferably, *semantic* meaning in the context of the task, for example the *point of a pencil*. Decré et al. [51] extended the framework to support inequality constraints.

A general iTaSC task is the *composition* of multiple sub-tasks, involving possible multiple *robots*, *sensors* and *objects*, and at the level of that composite task, *weights* and/or *priorities* between the sub-tasks can be introduced by the task programmer. This specification is then turned into a numerical *constrained optimization problem*, from which a *solver algorithm* computes the instantaneously best joint setpoints (e.g., joint velocities or accelerations) for the robot(s) at each moment in time, which are then sent to the lower-level actuator hardware controllers.

The key advantages of the "iTaSC paradigm" are: (i) a *systematic workflow* to define task constraint expressions [165]; (ii) the *composability of constraints*, since not only can multiple constraints be combined, but each of them can also be *partial*, that is, not constraining the full set of degrees-of-freedom (DOF) of the robot system or of the task space; (iii) *reusability of constraints*, since the (recent) DSL support allows to specify relation between object frames in symbolic form, hence with (potentially) more semantic and hence higher and more context-specific reusability; (iv) *derivation of the control solution*: the iTaSC approach systematically evaluates the task constraint expressions at run time and generates setpoints for a low-level controller; (v) *modeling of uncertainty*: it provides a systematic approach to model and estimate uncertainties.

iTaSC is not the only software framework available for complex robot task specification. Three similar frameworks (developed independently and during overlapping periods in time) are known to the authors:

- *TaskNets*: Finkemeyer et al. [62] developed a control architecture and a software framework for the execution of Manipulation Primitive nets, including the integration of on-line trajectory generation [97]. Recently Thomas et al. provided the LightRocks [156] DSL for skill based robot programming.
- The *Stack of Tasks* (SoT) [102] framework provides a dataflow approach to the "Generalized Inverted *Kinematics*" computations required in complex compositions of several sub-tasks for the robot, in which the relative contributions of each sub-task can be prioritized with respect to the others.
- the *Stanford Whole-Body Control* framework (SWBC) [139] implements a hierarchical control structure, on the basis of *full-dynamics* "solvers". Also SWBC allows to establish priorities among several sub-tasks.

The single underlying paradigm of all these frameworks is that they rely on *a set* of *compliance frames* or *task frames*. Each of the task frames represents part of the overall task specification (which we call `Tasks` in the remainder of this text), and adds a set of *objective functions* and *constraints* to a *solver* that then has to compute the "optimal" solution to the (possibly overconstrained or underconstrained) overall constrained optimization problem.

In contract to SoT and SWBC, iTaSC and TaskNets introduce some extra software in their framework, namely *Finite State Machines*, to specify and execute also the *discrete behavior*, that is, the sequencing of particular sets of sub-tasks (each of which specifies a continuous time/space behavior).

5.3.3 Relation to the chapter

All of the frameworks mentioned in Section 5.3 *have* paid attention to the *integration* challenge, but, invariably, this is still limited to "adding extra functionalities into our own framework", but not (yet) "integration of selected functionalities from different frameworks into the same application". Hence, the ambition of this chapter is to explain how to (re)design software frameworks, such that the latter type of real integration can be supported in a more maintainable way; here, the "maintainability" context is that of independent "third parties", and not that of the original developers of the framework. "Real integration" also means that the provided functionality can be used as building blocks in *any other* system architecture than the one(s) used by the original developers.

Many of the architectures discussed in Section 5.3 conform to a certain degree to the architectural Composition Pattern. However, none of these architectures is known to incorporate or separate all aspects of the pattern, explained in following sections, explicitly.

Moreover, the Composition Pattern does not limit the composition hierarchy to a fixed number of layers or tiers, nor to a hierarchy of "general to specific" layers of abstraction.

5.3.4 Lessons learned from refactoring the iTaSC framework

As mentioned before, the authors get their "best practice" insights mainly, but by far not exclusively, from the long-term development efforts of the *iTaSC* framework, [46], whose functionality is summarized in Section 5.3.2.

The first *iTaSC* software was developed by Ruben Smits [144], influenced heavily by the features available at that time in our other large-scale software framework *Orocos* [33]. Both frameworks were, in themselves, already improved (and "decoupled") versions of our previous-generation (too) highly integrated robot specification and control software framework *COMRADE*, which dates back to the early 1990s [159, 175]. Recently, Vanthienen et al. [167] created a second-generation iTaSC implementation, profiting from the "best practices" presented in this chapter; the major difference with the first generation is the higher degree of formalization and structure of the iTaSC paradigm, supported by the formal *domain-specific language* presented in chapter 4. Hence, a developer can create an iTaSC *model* of an application (instead of directly having to write the *code*), and that model is parsed, transformed into structured code templates, and then executed by a running instance of the *code framework* presented in this chapter. The higher degree of formalization, separation of

concerns, and the accompanying structure, enable developers to reuse tasks specified and implemented before in combination with other tasks to form a new application, and on any robot that can be represented by a kinematic tree. Moreover, it allows reuse on an even more fine-grained level of for example *only* the statechart ("Coordination", that is, the *discrete* behavior of a task). All this can now happen with much smaller configuration files that have to be changed during the reuse, compared to the first-generation version.

Examples of concrete limitations for reuse, adaptability, and extensibility, encountered in earlier work, which where solved using the "best practices" introduced in this chapter, include: (i) conditional statements (if-then-else) in components that in fact do scheduling or coordination of the component, for example combining the procedure to bring a robot to a running state, with the (general applicable) kinematic algorithms to calculate end-effector positions; (ii) interfaces that communicate data, which are in fact events; (iii) the coupling of application specific configuration and monitoring with the functional behavior, inside a component.

In summary, the presented "best practices" are grounded in the accumulated software development experiences of several dozen researchers spanning more than 20 years of very focused framework developments, and several generations and types of computational and robotics hardware.

5.4 Composition

Composition is the first of the *5Cs* to be discussed. It models the *structure* of the coupling between the entities of the other concerns; those other concerns (Computation, Configuration, Coordination and Communication) model four complementary kinds of *behavior* in a system. The structural model in the Composition deals with two aspects: on the one hand, it groups entities together in *composites*, supporting **hierarchy**, and on the other hand, it models the **interactions** between the system entities. Composition (or "architecture") is a trade-off between *composability*, i.e. the property of an entity to be easily reused in a Composition, and *compositionality*, i.e. the property of a composite to have predictable behavior knowing the behavior of its components [32]. To the best of the authors' knowledge, no scientific insights are known about how to optimize the architecture of complex systems; hence, Composition remains much of an art, while for the other C's described below, some more concrete design insights and guidelines do exist.

5.4.1 Modeling

Figure 5.1 shows the **pattern of composition**, one of the two major "best practices" presented in this chapter (together with the "5Cs"). The pattern forces developers to consider *any* composite entity as consisting of following entities:[‡] [§]

- *Functional Entities* (Computations) deliver the functional, algorithmic part of a system, that is, the *continuous time and space behavior*. A `Functional Entity` can be a composite entity in itself, following the same pattern of composition. Section 5.5 elaborates on (Composite) `Functional Entities`.
- A *Coordinator* selects the *discrete time behavior* of the entities within its own level of composition, that is, to determine which continuous time behavior each of the Functional Entities in the composite must have at each moment in time. Section 5.7 elaborates on `Coordinators`.
- A *Scheduler* handles the *resource access and timing* of the entities within a composite, e.g. the order of execution of the Functional Entities (computations) within the composite entity (including access to shared data), required for correct overall behavior of the composite. Section 5.7 elaborates on `Schedulers`.
- A *Communicator* handles *the exchange of data and events* between entities.
  - *Data communication* handles the *data exchange behavior* between Functional Entities and Monitors, Configurators, or other Functional Entities. Note that data communication is, in general, *bi-directional*, in contrast to the popular mainstream "publish-subscribe" tradition.
  - *Event data communication* handles communication between all entities and the `Coordinator`.

  Section 5.8 elaborates on `Communicators`.
- A *Monitor* compares the *actually* received and sent out data with *expected or reference* data, and fires events depending on a configurable set of constraint conditions that must be monitored for a robust execution of the composite. Section 5.5 elaborates on `Monitors`.

[‡]See also the definition in Section 3.2. In contrast to Section 3.2, this section explains the difference in entity types by the run-time behavior of their implementations.

[§]In some cases, it might make sense to eliminate one or more of these entities, but then, at least, the developer has a motivated reason to do so.

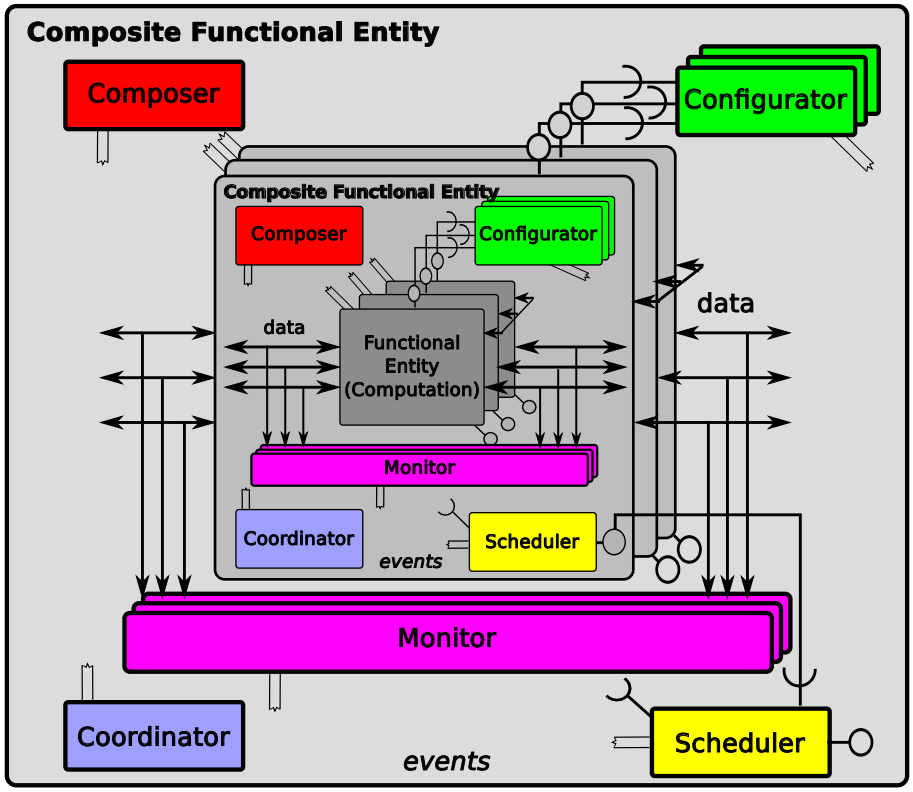


Figure 5.1: Pattern of composition. Each block represents an entity, arrows indicate data communication, double lines indicate event communication, and a line with the lollipop-socket indicate event or service providing-requesting. Since entities are broadcast, the double lines represent a 'bus system' and are only partially drawn. Three layered blocks indicate one or multiple entities of the same type. The `Composer` (red), `Coordinator` (blue) and `Scheduler` (yellow) are "singletons" within a Composite `Functional Entity` (grey) because they all are "master" of the (possible multiple) (Composite) `Functional Entities`, `Monitors` (purple) and `Configurators` (green), at different phases in the composite component's life cycle. Each `Functional Entity` can be (replaced by) a Composite `Functional Entity`, which leads to hierarchy of compositions. A hierarchy with a depth of three is shown in the figure; a darker shade of grey indicates a (Composite) `Functional Entity` at a deeper depth level within the hierarchy.

- A *Configurator* configures the entities within a level of composition. Section 5.6 elaborates on `Configurators`.

- A *Composer* constructs a Composition by grouping and connecting entities. This section further elaborates on `Composers`.

The composition pattern is recursively applicable (as suggested by Fig. 5.1), with each `Functional Entity` in each hierarchical level following the same composition structure. This gives the possibility of creating a hierarchy of large numbers of composite entities, without having to learn any new architectural design primitives, or adapt one's design trade-off insights. In other words, the authors' "best practice" suggests to use this composite pattern as the *smallest architectural building block*, which is in strong contrast to the more mainstream belief that the single entities (or "component") themselves are *the* most appropriate system primitives for composition or reuse. The impact of this difference on overall system architecture can not be overestimated, and hence it is a very important point for discussion and review. Again, this "best practice" has grown out, step by step, from the above-mentioned large body of software systems that have been built by the authors' research group, in isolation or in close cooperation with international partners. That means that the role of *each* of the parts in the pattern is motivated by several concrete use cases, in a multitude of application scenarios and contexts.

One successful, independently created instance of (a large part of) this composition pattern is realised in the *RObot Construction Kit* (ROCK) [80]. It was the first publicly available software project to introduce what this chapter calls the `Composer`, as a necessary entity within any composite. Its role is to group and connect all other entities, on the basis of a *model* of the architecture. Its first responsibility is the deployment of the entities within a Composite Functional Entity, when the system is brought alive for the first time. However, the `Composer` is *active throughout the whole life-time* of a Composite Functional Entity, and responsible for *run-time* changes in the system architecture. A `Composer` as an entity in its own right allows the `Coordinator` to trigger a (re-)composition of the Composite Functional Entity or a gradual composition, intermittent with configuration steps for the composed entities. This acknowledges the `Composer` as a real "activity" and not a static data structure.

The interaction between `Composer` and `Coordinator` follows a **`Coordinator-Composer` pattern**, a specialisation of the `Coordinator-Configurator` pattern introduced by Klotzbücher et al. [89]. In the `Coordinator-Composer` pattern, a `Composer` holds a set of composition steps. Each composition step has a unique ID and can be implementation or software specific. The `Coordinator` *commands* the composition steps to be executed, the `Composer` *executes* the commanded composition steps, that is, it is configuring the *structural* model of the Composition; the `Configurator` in a composite, on

the other hand, is changing the *behavior* of the composite but not its structure. Of course, changing the composite's structure most often implies that first a change in the composite's behavior must be realised, in order to bring the composite to a behavior that allows the restructuring.

This `Coordinator` commands in the form of raising events, on which a `Composer` reacts when the event matches a composition step ID. A status event communicates success or failure of the composition step back to the `Coordinator`, allowing a befitting reaction. Section 5.7 details the interaction between the `Coordinator`, `Composer`, and `Configurator`.

The `Functional Entities` take a special position within the pattern of composition: (i) there can be multiple `Functional Entities` within a Composition, and (ii) a `Functional Entity` can be a Composite Functional Entity in itself, following the pattern of composition of Figure 5.1, resulting in a hierarchy of composites.

`Functional Entities` take this special position since they form the core functionality of a system: without them the other entities have no meaning nor use. Moreover, the other entities exist *only because* the behavior and interaction structure of multiple `Functional Entities` need extensive "bookkeeping" support.

The `Functional Entities` described in Section 5.5 are grouped in a hierarchy of composites, further referred to as *levels of composition*. A *higher* level of composition, the parent, consists of a Composition of *lower* level components, the children. Section 5.5.2 elaborates on these levels of hierarchy.

The presented hierarchy of composition has the semantic content of a **boundary of knowledge**. The entities within a Composite Functional Entity only know about the presence of the other entities within that composite. This does not hold for the `Functional Entities`: each of them *should not know* about any of the other entities in the composite, since everything that has to be known is already covered in the other entities. Hence, the `Functional Entities` *broadcast* their data and events, not having to know who will react or use them. It is the authors' belief that this pattern represents the most strict decoupling between entities that still results in a manageable and comprehensive entity, composite and system design.

Figure 5.3 gives an example of this concept applied to an example `Task` in the context of the *iTaSC* framework. The `Coordinator` raises an `e_CC_PID_connect` event in its *ConnectEntities* substate. The `Composer` reacts to this event by creating, amongst others, a connection between the *Chif* ports of the `Functional Entities` *VKC_Cartesian* and *CC_PID*. Sections 5.5, 5.6, and 5.7 will further elaborate on this example.

5.4.2 Implementation

Composition as a concept *composes* entities of all the other 5C concerns; details of the latter will be given in their respective Sections.

The current implementation of `Composer` is a Lua [74] script using the RTT-Lua extension libraries [94]. The `Composer` scripts are loaded in an Orocos-Lua component [94]. An Orocos-Lua component provides a Lua based execution environment for constructing real-time safe robotic domain-specific languages. It gives the features of an Orocos component, such as Communication and Configuration infrastructure (ports, property marshalling) to a `Composer`.

The RTT-Lua extension libraries provide the software framework specific information to create and connect entities, in this case *deploying Orocos components* and *connecting Orocos ports*. As will be elaborated in dedicated sections, all entities will be deployed in an Orocos component.

The implementation provides a boiler plate script for the `Composer` for the default compositions made in iTaSC (see Section 5.5.2), and this is possible because of the very fixed structural model to which the involved implementations of the entities conform. Future work will create a domain-specific language for the `Composer` in line with the `Coordinator` DSL [89], Figure 5.3 hints at such an implementation.

The reference iTaSC framework implementation groups code related to a Composite Functional Entity in a ROS package. For example such package contains the C++ code for the `Functional Entities` and `Monitors`, rFSM/RTT-Lua Lua scripts for the `Coordinators`, `Configurators`, `Composers`, and `Schedulers`, XML property files for the `Configurators` and references (e.g. ROS dependencies) to leaf composite entities.

5.4.3 Discussion and lessons learned

In a first implementation the `Configurator`, `Coordinator`, and `Composer` were loaded in a single Orocos-Lua component. The advantage of this approach was the shared activity (thread) and memory, reducing the need for event communication; also the human factor was important: at that time, we worked in a context where typically one single developer was responsible for most phases in the development process, so it was the easiest solution for this single developer to put all configurations, deployments, and coordinations into one single file.

This simple approach turned out to have severe disadvantages in the longer

term: the sharing of activity and memory implicitly also causes the coupling of these entities, because the blocking of an operation in a `Coordinator` or `Composer`, causes the thread to block, leaving possible identification and reaction only to the higher level `Coordinator`. The latter typically can do no more than identify that the whole Composite Functional Entity has stalled. The current separation of the entities as single Orocos components conforms better to the separation of concerns advocated in this chapter.

Another "lesson learned" in this context was about the human factor: large configuration files make it *extremely difficult* for new developers (i) to understand the whole file, and, hence, (ii) to be confident that they understand the implications of whatever small change they would like to make to the configuration file. In practice, this had led to very poor reuse of existing code, and even too close to zero incremental improvement of the existing code.

From the "component" framework point of view, we learned that it is impossible to create something like a generic default script for a `Composer`, since Orocos-RTT (or ROS, or any other "component" framework) lacks an explicit, let alone formal, model of components and of how they can be composed. However the above-mentioned *ROCK* project [80], which builds on Orocos-RTT, has made very good steps at bringing in such formal modeling for Orocos components and their composites, via its *Syskit* sub-project.

From the "task specification" framework point of view, none of the frameworks mentioned in Section 5.2 provides hierarchy for their *software entities*; many do offer hierarchy for their *task specification* primitives, but this hierarchy has very different purposes. A task specification (in a model-driven engineering context) is a formal description (model) of what the robot system should do. Hierarchy has been introduced in that context since basically the beginnings (early 80s), in the form of more or less detail in the task description; for example the task of navigating from Room_A to Room_B in a building is hierarchically decomposed into the subtasks of navigating (i) within Room_A from the robot's current position to the door of Room_A with Corridor_1, (ii) through Corridor_1 to Corridor_2, (iii) inside Corridor_2 to the door of Room_B, (iv) from the door of Room_B to the desired end location inside Room_B. And each of these sub-tasks can be hierarchically decomposed in more fine-grained sub-sub-tasks, such as (i) moving the robot arm to the handle of the door in Room_A, (ii) grasping the door handle, (iii) turning the door handle crank, (iv) turning the door around its hinge, (v) releasing the handle grasp, (vi) moving the arm in a minimal-width configuration, (vii) moving through the door opening into Corridor_1. Etc. The hierarchy described above is 'orthogonal' to the software architecture hierarchy which is the focus of this chapter.

5.5 Computation

Computation (a `Functional Entity`) delivers the useful **functionality** of a system, i.e., the algorithmic part of an application. As mentioned above, applications typically involve *many* different `Functional Entities`.

5.5.1 Modeling

A task specification application, based on constraint-based programming according to the iTaSC approach, consists of the following `Functional Entities`:

- *Sensors* deliver *feature measurements* derived from raw sensor data, e.g., distance information, force-torque data, or point clouds.
- *Robots and Objects* calculate the state of robots and objects involved in an application based on their kinematic and dynamic models. `Robots` have controllable degrees-of-freedom (DOF), whose state is denoted with coordinates $\boldsymbol{q}$. Unlike `Robots`, `Objects` have no controllable DOF; their models comprise definitions of *object frames* as reference frames for state calculations such as the pose or twist between two object frames. Computations by `Robots` and `Objects` include forward and inverse kinematics and dynamics solvers, as implemented by for example the Orocos KDL library [146].
- A *Scene* provides the environment in which the `Robots` operate and in which the `Objects` are located.
- *Actuators* deliver hardware interfaces for `Robots` and `Objects`, communicating proprioceptive information, desired low-level controller setpoints, and sensor or estimator information. Examples include the Kuka FRI interface [138] or an interface to a controller provided by the pr2_controller_manager on a PR2 robot [173].
- A *World Model* keeps track of the position of the robots and objects in the scene, and between which object frames tasks are defined. It transforms data to be composable conforming to geometric semantics [42, 43], e.g., common reference frame and point, as well as object and reference object on which these are defined for the sum of poses.
- A *Solver* calculates the desired values for the low-level robot controllers as the result of the *constrained optimization problem* that results from the methodological composition of task constraints and objective functions.

Examples include mathematical optimization algorithms such as frequently used weighted-damped least-squares, or more complex algorithms provided by general-purpose numerical solver toolkits, e.g., ACADO [73].

- A *Task Space Representation* calculates the state of the task space or feature space defined between object frames of the robots and objects. It uses a kinematic model of this task space using the auxiliary feature frames. In its explicit form it can be regarded as a virtual kinematic chain (`VKC`) which state is represented by the *feature coordinates* $\boldsymbol{\chi_f}$ ("*Chi-f*"). Computations by `TSRs` or `VKCs` include forward and inverse kinematics and dynamics solvers.
- A *Constraint-Output (CO)* calculates the output equation $\boldsymbol{y} = f(\boldsymbol{\chi_f}, \boldsymbol{q})$. The output can serve as input for controllers, estimators, monitors etc.
- A *Constraint-Controller (CC)* calculates the control law that enforces a desired setpoint on an output, resulting in the desired output in task space, e.g. $\dot{\boldsymbol{y}}_d^\circ$ for the velocity resolved case. Examples of `Constraint-Controllers` include the commonly used PID controller or impedance controllers.
- *Setpoint generators* deliver desired values for the controllers of a `Constraint-Controller`. `Setpoint Generators` can provide *fixed values*, but also more *complex data structures*, or even full trajectory generating or planning *functions*.
- An *Estimator* observes or estimates the (internal) state of a system, based on a model, and the input and output of the system under observation. `Estimators` are commonly referred to as *state observers* in control theory, or an implementation of *adaptation* in computer science.

The *Composition Pattern* discussed in Section 5.4 introduces the *`Monitor`* as an essential, special Functional Entity. It compares the actual data flow between the `Functional Entities` with reference data, and raises an event when a configured set of conditions is met. For example, the `Monitor` on an `Estimator` that outputs the uncertainty on an estimated parameter, can raise an event to indicate that the uncertainty has risen above a maximum value; the composite's `Coordinator` can then react to that event by, for example. slowing down the current movement of the robot. (Event processing is discussed in detail in Section 5.7.)

Figure 5.3 shows an example of the interactions of a `Monitor` of a `Task` Composite Functional Entity. The `Coordinator` raises an `e_monitor_max_position_error` event that triggers the `Monitor` to monitor the position error. The `Monitor` has a connection to the data flow

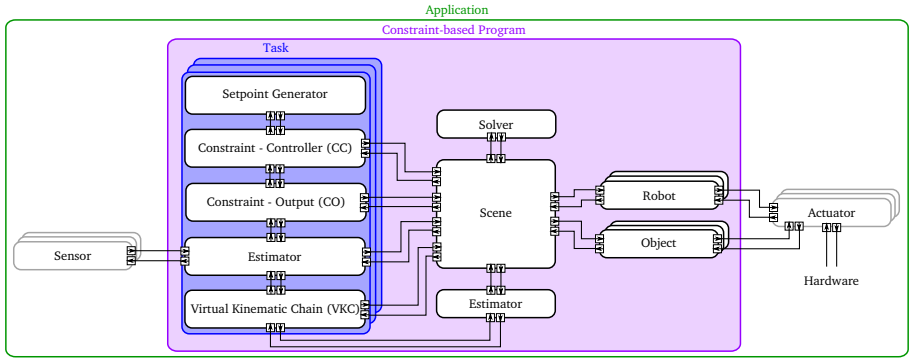


Figure 5.2: Detail of the composition of computation (Functional Entities) for the explicit formulation of iTaSC using sysML flow ports [122]. The composition levels are the *Application* context, the *Constraint-based Program*, and the *Task* specification. Stacked boxes refer to the possibility of having multiple entities of a specific type.

of the `Functional Entity` *CC_PID* that outputs this error. The `Monitor` raises the `e_max_pos_tracking_error_exc` event to indicate that the maximum allowed position tracking error has exceeded.

The *separation* of `Functional Entities` from their `Monitors` *decouples* Functional Entities and application specific monitoring conditions, resulting in higher reusability. Obviously, `Functional Entities` should also raise additional events themselves, based on *internal* monitoring conditions, such as the completion of a certain algorithm or the reaching of a maximum number of iterations.

5.5.2 Composition of `Functional Entities`

The following paragraphs describe the *levels of composition* for the use case of constraint-based optimization. We restrict ourselves to three levels of composition, *Application*, *Constraint-based program*, and *Task*. Higher or lower level composites are definitely possible, for example the higher level of a *Mission* that incorporates multiple applications, deployed simultaneously, or serially on multiple robots. At each of the three levels we focus on, we regard the most important example of a composite, which also gives its name to the level. This does not limit the other Functional Entities to be composites following the pattern of composition. For example a `Sensor` can be a composite of an `Actuator`, a `Filter`, and other algorithms on the sensor data, possibly divided

over multiple levels of composition. Or a `Constraint-Controller` can consist of a Composition of atomic controllers for each of the degrees-of-freedom of the output equation. The structure of the Communication, Configuration, and Coordination on each level will be discussed in their respective sections.

- A *Task* forms a first composite, as shown in Figure 5.2, delivering a set of constraints that are related to the same task space to the optimization problem[¶]. In case of the explicit formulation of iTaSC, a `Task` composes the `Virtual Kinematic Chain (VKC)`, a `Constraint-Controller (CC)`, the `Constraint-Output (CO)`, and a `Setpoint Generator` as shown in Figure 5.2. This composite contains all functionality needed to define and execute a task. This `Task` is however agnostic of its concrete role in the whole application. For example, it is unaware of the role or context of the object frames between which it is acting. That semantic meaning is (or rather, should be...) given by the parent composite. The current discussion limits itself to one entity of each type, however multiple entities can be present to be able to switch between `Constraint-Controllers` or `Setpoint Generators` within one `Task`, or entities can be brought out of the `Task` composite. This discussion is beyond the scope of the current document.
- A *Constraint-based Program* forms a second composite, as shown in Figure 5.2, delivering task specification and control on a set of involved robots and objects. This Composite Functional Entity composes all elements related to the *World Model* and how it is used to generate setpoints for the low-level (motor) commands. It composes ("couples") the `Robots` and `Objects` in a `Scene`, constrained and linked by `Tasks` which encompasses the task space formulation and resolved by a `Solver`.
- The *Application* forms our third composite, composing ("coupling") the `Constraint-Based Program` with the application-specific "hardware" (`Platform` with its `Actuators` and `Sensors`), as shown in Figure 5.2. Separating the hardware from the program allows developers to reuse the same `Constraint-Based Program` in simulation or on the real robot by just changing the `Actuator` and `Sensors`, and offers flexibility with respect to the hardware used.

As mentioned in Section 5.4, a Composition forms a boundary of knowledge. The following example explains this concept; the `Configurator` named *iTaSC_Configurator* of a `Constraint-Based Program` named *iTaSC*

[¶]Section 2.2.2 gives an overview of constraint-based programming and iTaSC related terminology.

needs to know which `Tasks` to configure, and the `Coordinator` named *iTaSC_Coordinator* needs to know which `Tasks` to expect events from. The `Tasks` however present their data on a data flow port, not knowing who is using the data. It is the `Composer` of *iTaSC* that determines who is listening and reacting. For example the `Monitor` named *iTaSC_Monitor* that monitors the data of a specific `Task` named *ApproachObject*. In addition to the *(functional) data*, the *ApproachObject* also broadcasts *events*, and it is the *iTaSC_Coordinator* that expects and reacts on events from the *ApproachObject*.

5.5.3 Implementation

The model provided above is implementable with various software component frameworks or their combination, such as OpenRTM [7, 114], Orca [27], GenoM [65] or ROS [130]. Strictly speaking, the Composition Pattern requires only the following primitives to be provided by software frameworks: Component, Port, DataFlow, Event, FiniteStateMachine. All these primitives are provided by many frameworks, but no framework provides them all; except for ROS or OpenRTM, when the definition of *framework* is taken in the broader sense of *original framework and the ecosystem that grew around it*. However, it is not at all necessary that an implementation of the Composition Pattern has to be realised in one single framework; on the contrary, the 'best' implementation will most often consist of a selection of features from different frameworks. Of course, 'best' is an application-specific objective function, and sometimes 'real-time performance' will be part of that objective function (making the Orocos framework more appropriate than ROS, for example), while another application gives less weight to real-time performance than to the desire to reuse already existing ROS node implementations,

Our reference implementation provides two different approaches to implement `Functional Entities`. The first and most often used approach is applied throughout the core of the implementation and uses the Orocos component framework for real-time control [29, 33, 149] to provide an infrastructure for `Functional Entities`.

The model of an Orocos component has three primitives: Operation, Property, and Data Port. That means that in the (semantically rather restricted) context of component frameworks, it is the component that provides the basic unit of computational functionality. Data needed for calculations and the resulting data of a component's calculations is communicated using Data Ports.

The advantages of the Orocos components as `Functional Entities` include: 1. the real-time capabilities, 2. thread-safe time determinism, 3. lock free inter-component communication in a single process, 4. synchronous and asynchronous

communication possibilities, 5. reflection capabilities and interfaces to other frameworks such as ROS. Their disadvantage is that most developers (implicitly and incorrectly!) assume that each component has to be deployed in its own operating system process, but this policy of composition introduces many *context switches*, most of which are functionally superfluous.

The Orocos component framework does not *explicitly* provide composite components. However, since the software patterns presented in this chapter offer *composition by infrastructure* (Section 5.4), this lack of explicit Orocos composite components is not a fundamental problem to the formalization of `Functional Entities` as composite entities.

In order to ensure modularity and reusability of the components as instances of `Functional Entities`, they have to provide a well defined Data Port interface: what data should be communicated and in which form. (Section 5.8 elaborates on the communication aspects of these issues.) Therefore the reference implementation offers a template component for each type of `Functional Entity`, in the form of a C++ class. More specialised components inherit from this template.

For example a `PID` or `impedance_control` component inherits from the `Constraint-Controller` template, implementing a PID controller and impedance controller respectively. Both components are however still *general* in the sense that their behavior will depend on

- their composition and communication that determines who delivers setpoints and state information,
- their configuration that determines which gains to use,
- their coordination that determines when they are active.

A component can also serve as an interface to other parts of software or hardware, for example a `Actuator` that interfaces with a KUKA robot over an FRI connection [138].

In addition to Orocos components that inherit from a template, the reference implementation provides a second, more general way of introducing `Functional Entities` by adding meta-data to an implementation of a `Functional Entity`. This meta-data models the interface of the `Functional Entity` and contains the necessary information for other entities to interact with the entity. Listing 5.1 gives an example of meta-data of the Data Ports of a `Functional Entity` using the *Lua* language [74]. It contains an entry in the Lua table for each Data Port by its name with following tags:

- type: the type of the port that defines what general kind of information the port delivers or requests,
- rtt_type: the type of the data specific to different platforms, in this case Orocos RTT,
- semantics: the (geometric) semantics meaning of data, the importance which will discussed in Section 5.8,
- id: detailed identification of the port, this could refer to for example ROS topic information,
- direction: the direction of the Data Flow with respect to the entity,
- fw: framework in which the entity is implemented, which will define how to interpret the other tags such as the id.

The reference implementation uses this meta-data approach to ,for example, provide a `Actuator` for the KUKA Youbot using the existing open-source ROS nodes provided by *youbot_description*. [99].

Listing 5.1: Example of meta-data

```
ports={
  my_port_a={rtt_type='/motion_control_msgs/JointVelocities',
    type='driver',
    semantics='JointVelocitySemantics(ee,base)',
    id='/JointVelocitiesCommand',
    direction='input',
    fw='ros'}}
```

Our reference implementation implements `Monitors` for example using the service plugin feature of Orocos. It allows pluging in extra functionality in an existing Orocos component. Future work will formulate a DSL for `Monitors`, as hinted at in the `Monitor` entity shown in Figure 5.3.

5.5.4 Discussion and lessons learned

The majority of Functional Entities (computations) in the current implementation are encapsulated in the Orocos components, which mirrors the proposed `Functional Entity` model. However, this structure of different components with (inter-process) communication is also very rigid with respect to optimization of the computational efficiency. Nevertheless, models can be deployed in different ways. For example, certain Functional Entities could be grouped at run-time,

reducing communication needs. An example of such composition can be found in the GenoM project, that makes use of codels [65] as the smallest unit of execution that can be easily composed to larger Functional Entities without inter-process communication, for example using shared memory.

The approach to attach a model to an implementation gives more versatility. The current implementation gives only a limited example of such an approach.

5.6 Configuration

Configuration influences the behavior of entities of the other concerns by changing its **settings**. Examples include control gains and communication buffer sizes.

5.6.1 Modeling

Configuration is enforced by a `Configurator` entity, separating it from coordination by the `Coordinator-Configurator` Pattern [89]. A `Configurator` holds a set of configurations.

A configuration consists of a set of parameters of another entity that are exposed to be configurable. It has a unique name and can be implementation-, hard- or software specific.

The `Coordinator` *commands* the configurations to be loaded in an entity, the `Configurator` *executes* the commanded configuration. A `Configurator` applies a configuration with a certain name when receiving an event from the `Coordinator` with a matching ID. A status event communicates success or failure of the configuration action back to the `Coordinator`, allowing a befitting reaction.

Figure 5.3 gives an example of the `Coordinator-Configurator` interaction for an example `Task` Composite Functional Entity. The `Coordinator` commands a high tracking accuracy of a controller by raising an event *e_high_accuracy_control*, on which the `Configurator` reacts with adapting the gains of the `Functional Entity` *CC_PID* to a preset value.

Another example is the configuration of a `Monitor`, as also shown in Figure 5.3. The `Configurator` configures the `Monitor` with the concrete error level to react on, and the resulting events to raise, in this example *e_max_pos_tracking_error_exc*.

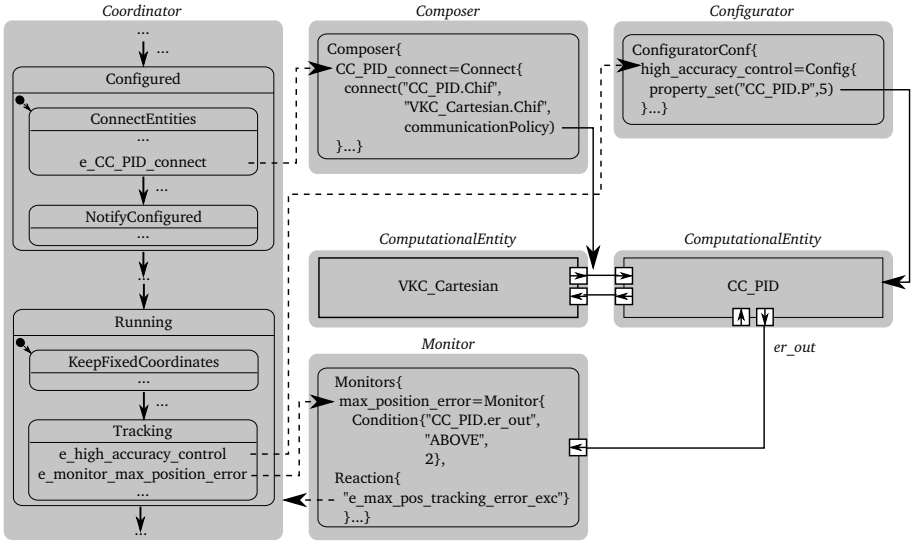


Figure 5.3: Example of the interaction of the `Coordinator`, the `Composer`, the `Configurator`, and `Functional Entities` of an example `Task` Composite Functional Entity. Dashed arrows indicate how events trigger actions, black arrows indicate how entities act on other entities. Only the parts relevant for the example are shown, the `Scheduler` and other `Functional Entities` are left out. Three dots indicate left out parts within an entity.

The `Configurator` needs to be configured itself, which seems a contradiction at first glance. It is however the hierarchy provided by the composition that allows the configuration of the `Configurator`: The `Configurator` of the level of composition higher will configure the `Functional Entity` to which this `Configurator` belongs to. This configuration includes the configuration of this `Configurator`. For example the `Configurator` *iTaSC_Configurator* of a `Constraint-Based Program` *iTaSC* configures a `Task` *ApproachObject*, hence configuring its `Configurator` *ApproachObject_Configurator*. A bootstrap ensures the configuration of the `Configurator` of the highest level Composite Functional Entity. Section 5.7 details the bootstrap to bring up the system.

5.6.2 Implementation

Since the reference implementation mainly uses Orocos, its Property infrastructure is used for configuration. Orocos Properties [149] provide an interface to

adapt at run-time parameters that are made publicly available. Services to read and write these properties to XML, RTT-Lua or other formats are available for the Orocos platform. Configuration specified in the iTaSC DSL or deduced from it can be set accordingly.

The `Configurator` implementation uses the reference implementation of the Coordinator-Configurator pattern in Lua. Its extension with the RTT-Lua libraries provides the software framework specific information to configure Orocos components.

As for the `Composer`, the implementation provides a boiler plate script for the `Configurator` for the default compositions made in iTaSC. The configuration of the `Configurator` can load different sets of configuration. For example the configuration of a `Configurator` of a `Constraint-Controller` comprises the loading of the gains stated in the iTaSC DSL model (the configuration) into the `Configurator`, which applies the correct set of gains on the instance of the `Constraint-Controller` on receiving a command from the `Coordinator`.

5.6.3 Discussion and lessons learned

As for the `Composer` detailed in Section 5.4.3, separating the `Configurator` from the `Coordinator`, relieves the `Coordinator` from software platform specific actions and decouples execution and hence failure of `Coordinator` and `Configurator`.

5.7 Coordination

Coordination determines how the entities of all concerns work together, by selecting in each a certain behavior. It provides the **discrete behavior** of entities and their composites.

5.7.1 Modeling

Each Composite Functional Entity has one `Coordinator` that interacts with entities of other concerns by events. The model of the `Coordinator` is a rFSM statechart, introduced by Klotzb$\ddot{u}$cher and Bruyninckx [91]. Statecharts have the advantage to be composable, moreover rFSM statecharts are able to satisfy real-time constraints. The extended version of rFSM includes event memory, the use of which will be explained further on.

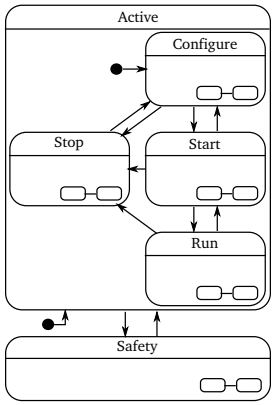


Figure 5.4: Life-cycle coordination pattern. The Active state consists of a Configure, Start, Run, and Stop state. The Safety state next to the Active state allows transition to this Safety state at highest priority. Each state can be a state machine of its own indicated with the two connected ovals in the right corner of a state. Figure 5.5 gives an example of the substates. The arrows indicate a state transition which is triggered by an event, the filled black circle indicates an initial connector.

The model of the `Coordinator` follows the best practice of *pure coordination* [91]. Pure `Coordinators` are *event processors*, which, as only functionality, determine states (based on events) and send out events. Pure coordination avoids dependencies on platform specific actions, and avoids blocking invocations of operations. The events originate from the other entities of a Composite Functional Entity or from a `Coordinator` of a parent or leaf entity.

A `Coordinator` conforms to the **life-cycle FSM** of a Composite Functional Entity, as represented in Figure 5.4. Each of the states of the life-cycle FSM can be a state machine on its own, hence a `Coordinator` is a *hierarchical FSM*. The following states are part of the life-cycle FSM:

- The *Active state*, which consists of the Configure, Start, Run, and Stop state. This state is the initial state when a `Coordinator` is brought up, indicated by the outer initial connector in Figure 5.4.
- The *Configure substate* coordinates the composition and configuration of the Composite Functional Entity. In the Configure state the `Coordinator` triggers the `Composer` and the `Configurator`. The `Composer` composes the entities of the Composite Functional Entity by creating entities (deployment) and connecting communication channels

between entities, as explained in Section 5.4 and 5.8. The `Configurator` loads and executes the configuration of all entities of the Composite Functional Entity, following the `Coordinator-Configurator` pattern [89] as explained in Section 5.6. The `Coordinator` triggers the `Composer` and the `Configurator` intermittent, since some steps of the composition need prior configuration. For example the creation of communication ports of the `Scene` dependent of the number of `Tasks` (configuration step), which can only be connected after their creation (composition step). This substate is the initial substate when a `Coordinator` is brought up, indicated by the inner initial connector in Figure 5.4.

- The *Start substate* coordinates the preparation of the entities of the Composite Functional Entity for nominal operation. In the Start state the `Coordinator` triggers the `Scheduler` to initialize, and Functional Entities to start computation and data exchange.

- The *Run substate* is the state of nominal operation of the Composite Functional Entity. On the one hand, the `Coordinator` triggers when entering this state the activation of the `Scheduler`. On the other hand, it influences the run-time behavior of the Composite Functional Entity. This run-time behavior consists of altering the active set and configuration of `Functional Entities`, based on incoming events fired by for example the `Monitor`. For example the configuration of the `Constraint-Based Program` consists of amongst others, the set of active tasks, the involved objects and (parts of) robots, and the task weights and priorities.

- The *Stop substate* coordinates the termination and destruction of the entities of a Composite Functional Entity. In this state, the `Coordinator` triggers the `Configurator` to do the 'opposite' of the actions during the Configure state.

- The *Safety state* brings the composite state in a safe mode, which does not necessarily correspond with the Stop state. Since the Safety state is located on a higher level in the state machine hierarchy, events triggering a transition to the Safety state will always have priority, independent of the current substate within the Active state. For example blocking the motors of a robot in an application in which the robot has to handle dangerous materials, or on the contrary, bringing the robot to gravity compensation mode when working close to humans. As the outer initial connector indicates, recovering from a Safety state requires a reconfiguration.

The different `Coordinators` over the different composition levels interact by events, forming a *hierarchy of concurrently executed (hierarchical) FSMs* in which

the higher level Coordinator coordinates the lower level Coordinators. Remark that not all Coordinators need to be in the same state, for example when a new Task is added to an existing Constraint-Based Program in the Run state and needs to go through the life-cycle until the Run state.

The life-cycle FSM takes part in the deployment of the system. A bootstrap brings up the highest level Composer that deploys the Coordinator and communication between Composer and Coordinator. Further this bootstrap ensures the configuration of the highest level Configurator. The Coordinator brings up the remainder of the Composite Functional Entity by a coordination of a series of composition and configuration steps, using the Coordinator-Configurator [89] and Coordinator-Composer pattern, explained in Section 5.4 and 5.6.

The advantages of this approach of deployment are (i) the systematic approach to bring up a system, (ii) the reduction of the actual phase of bringing up the system to a 'minimal' bootstrap, by using the structure of the composition, (iii) the predictability of the deployment procedure and its possible errors.

In addition to the Coordinator, the Scheduler forms part of the coordination. The Scheduler handles the order of the computations by the Functional Entities. However it forms not part of the Coordinator, since (i) a Scheduler uses often service calls or events, therefore it is not a pure event processor, (ii) a Scheduler forms a periodic (time-triggered) process with respect to the (mostly) aperiodically, event triggered behavior of the Coordinator, (iii) a Scheduler depends on the implementation of the Functional Entities, (iv) a Scheduler must be fast and efficient, and can therefore be optimized using specialized routines. Scheduler and Coordinator are separately triggered by their counterpart at a higher level of composition. This separation avoids coupling of the timing of Scheduler and Coordinator, that could cause delays in the scheduling. For example the Scheduler *iTaSC_scheduler* of a Constraint-Based Program *iTaSC* triggers a Functional Entity *Task* that itself is a Composite Functional Entity, this *Task* should immediately execute the algorithm, hence trigger its own Scheduler *Task_scheduler* and not wait for its own Coordinator *Task_coordinator* to command the *Task_scheduler* to do so. This avoids the situations where (i) the *Task_coordinator* has to react on both an event causing a behavior change and the trigger from the *iTaSC_scheduler*, (ii) and the situation where the *Task_coordinator* only reacts on the trigger from the *iTaSC_scheduler* in a next timestep (Section 5.7.2).

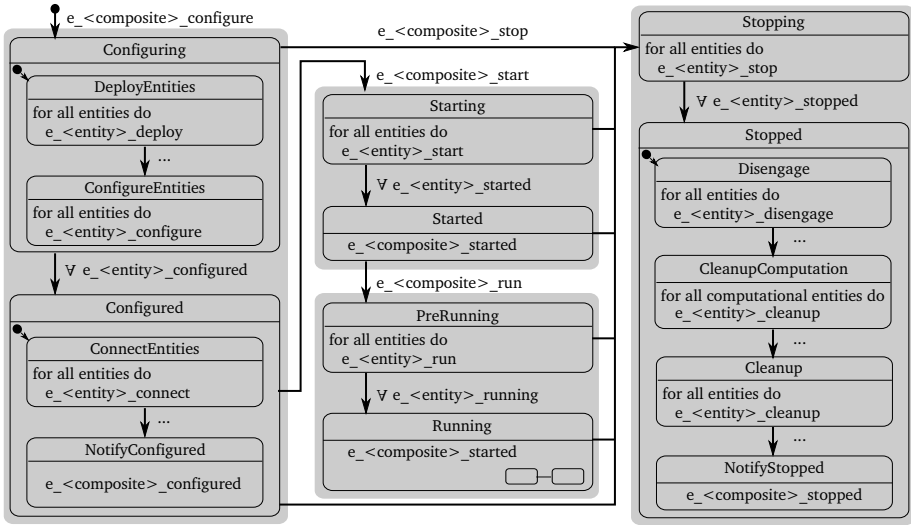


Figure 5.5: Detail of the Active state of the life-cycle FSM with example events. The arrows indicate a state transition which is triggered by an event, the filled black circle indicates an initial connector. Names starting with 'e_' denote events. Events next to arrows indicate the events that a state is waiting for to make that transition, events within a state indicate the events sent out by the state. The ∀ symbol denotes that all events of that type need to be raised to make that transition. < *entity* > denotes a name of an entity within the composite, < *composite* > denotes the name of the Composite Functional Entity this `Coordinator` belongs to. The grey background denotes the substates of the Active state as shown in Figure 5.4. Events that trigger the lowest level transitions are replaced by ... for readability. Also returning transitions such as from PreRunning to Started are left out for readability.

Concrete model of the life-cycle FSM

Figure 5.5 shows the details of the substates of the Active state of the life-cycle FSM. The grey boxes on figure 5.5 indicate these substates: Configure, Start, Run and Stop. They have each two substates: one with a name ending on *-ing* and one with a name ending on *-ed*, with exception of the Run state which has a PreRunning and a Running substate for linguistical reasons.

When in a *-ing* state, composite entities are coordinated, before triggering the `Coordinators` of its child entities. When in a *-ed* state, composite entities are coordinated after the `Coordinators` of the child entities are triggered but before the parent `Coordinators` are notified with an event. The Composite

Functional Entity will be further on referred to as the composite.

A parent `Coordinator` triggers the transition to an *-ing* state, hence the name of the composite that the `Coordinator` belongs to is in the event name. A `Coordinator` transitions from an *-ing* state to an *-ed* state when triggered by events from the child `Coordinators`. Due to this hierarchy the Active substates consist of exactly two states.

For example within the Configure state of a `Constraint-Based Program` *iTaSC* that has two composite child entities: the `Tasks` *ApproachObject* and *AvoidObstacle*, as shown in Figure 5.6. The events `e_ApproachObject_configured` and `e_AvoidObstacle_configured` raised by their respectively `Tasks` trigger the transition from the Configuring state of the `Constraint-Based Program` *iTaSC* to its Configured state.

Another example is shown in Figure 5.3 and was detailed in previous sections.

Transitions that require events from multiple child `Coordinators` require the event memory extention of rFSM in order to avoid synchronization problems. An event is in the rFSM model an edge triggered event that lives only at that time instant. The event memory extension registers all events that could trigger a transition from the current state, starting from the moment the state was entered. In other words the event memory is cleared with every new state that is entered.

The following paragraphs give an overview of the function of each substate:

- The Configuring state consists of two substates: DeployEntities and ConfigureEntities. The first triggers the `Composer` to create the entities within the composite. The `Composer` will also enable event flow between the entities. The creation of child composite entities consists of the creation of its `Coordinator`, `Configurator`, and `Composer`, similar to the execution of the bootstrap to bring up the root Composite Functional Entity as mentioned in Section 5.7.1. A status event from the `Composer` triggers the transition to the second substate. The ConfigureEntities substate triggers the `Configurators` to configure the entities within the composite and the `Coordinators` of child composite entities to transition to their Configuring state.
- The Configured state also consists of two substates: ConnectEntities and NotifyConfigured. The first connects the data flow between the entities of the composite. This connection is made after the configuration of the child composite entities, since connections can be configuration dependent. The NotifyConfigured substate notifies the completion of the configuration step to the `Coordinator` of the parent Composite Functional Entity.

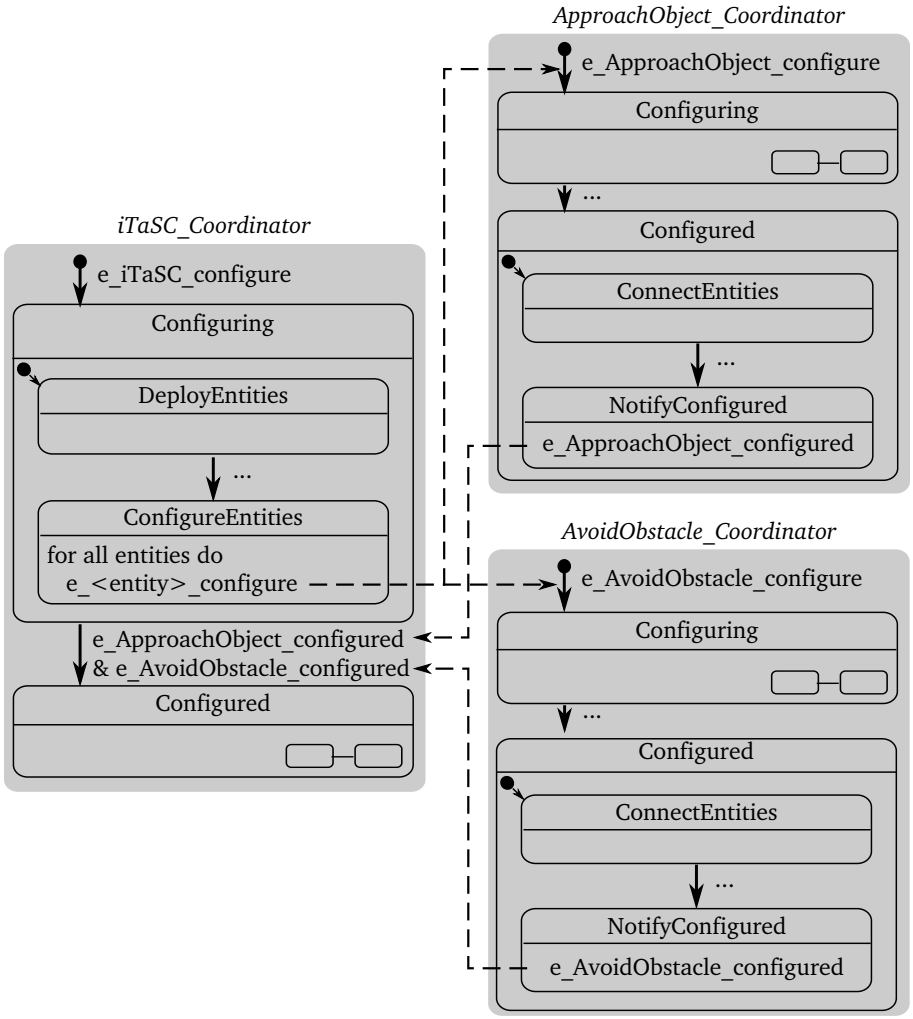


Figure 5.6: Example of the interaction between `Coordinators` at different levels of composition. The dashed arrows indicate how the raised event triggers a transition.

- The Starting state triggers the `Scheduler` to initialize, `Functional Entities` to start computation and data exchange, and triggers the `Coordinators` of child composite entities to transition to their Starting state.

- The Started state has as only function the notification to the `Coordinator` of the parent Composite Functional Entity.
- The PreRunning state triggers the `Coordinators` of the child composite entities to transition to the PreRunning state. It further triggers the activation of the `Scheduler`.
- The Running state notifies the `Coordinator` of the parent Composite Functional Entity and coordinates the run time behavior as explained in Section 5.7.1.
- The Stopping state has as only function the triggering of the `Coordinators` of child composite entities to go to the Stopping state.
- The Stopped state consists of four substates: Disengage, CleanupComputation, Cleanup, and NotifyStopped. The disengage substate triggers shutdown procedures, for example locking robot axes. The cleanup phase consists of two steps: CleanupComputation and Cleanup. The CleanupComputation state triggers the destruction of `Functional Entities`, including child composite entities. The Cleanup state triggers the destruction of the other entities within a composite. This distinction of two states allows the `Coordinator` to react on problems when destroying the `Functional Entities` for which it needs the other entities within a composite. In the last substate, NotifyStopped, the completion of the stopping is notified to the `Coordinator` of the parent Composite Functional Entity.

The execution of this pattern of coordination requires a model that provides the information of all separate parts and their relations. The iTaSC DSL [167] is an example that provides such a model.

The interaction of `Coordinators` outlined in previous paragraphs, details interaction in case of the existence of a parent Composite Functional Entity to the composite under consideration. The `Coordinator` of the root Composite Functional Entity will transition from a *-ed* to *-ing* state after completion of the latter, not triggered by an event of a parent `Coordinator`. The same structure applies to all entities, also for example to the `Actuator` and its sub-entities that coordinates robot hardware co-operation when composing different hardware.

5.7.2 Implementation

The iTaSC software framework uses the Lua reference implementation of the rFSM DSL for the `Coordinator`, that conform to the models presented

in previous sub-sections. These rFSM models are loaded in an Orocos-Lua component as for the `Composer`, `Configurator` and `Scheduler`, providing Communication and Configuration infrastructure to the entity. This component will be named *Supervisor* further on.

A *Supervisor* exposes the events raised within a `Coordinator` to an Orocos port, this port is connected to the other entities within a composite by the `Composer`. Through other Orocos ports, the *Supervisor* and hence `Coordinator` receives events.

The implementation considers two types of events, related to the state machine progression of the `Coordinator`: 1. *common events*, which are processed at each update of the `Coordinator`, 2. *priority events*, which are processed upon receiving them.

Most events are common events. Priority events are mainly used for 1. timer events, sent out by a periodic *Timer* to the root Composite Functional Entity, 2. events sent out by a `Scheduler` to trigger a `Functional Entity`, or its child `Scheduler` when that `Functional Entity` is a composite, 3. events that signal a fatal error, such as an `e_emergency` event. Hence a `Coordinator` is a hybrid event-triggered and time-triggered system.

The *Timer* triggers the `Scheduler` of the root Composite Functional Entity, which triggers his leaf `Schedulers`, which on their turn trigger their leaf `Schedulers` etc.

The implementation provides a boiler plate script for the life-cycle FSM, which is a general model and allows 'plugging in' the application specific part of the Running sub-state machine. These application specific parts can be developed and saved as separate rFSM models and hence files.

As mentioned in the modeling Section, a `Coordinator` knows the other entities within a composite, this knowledge is provided by the configuration of the `Coordinator`, derived for example from the iTaSC DSL model.

The current implementation of the iTaSC software framework provides a basic `Scheduler`, that requests operations on Orocos components in an algorithmic correct order with respect to the iTaSC concept.

5.7.3 Discussion and lessons learned

As detailed in previous sections, separating the `Configurator` and `Composer` from the `Coordinator`, leaves the `Coordinator` with no software platform specific actions, and is hence reusable with any other framework.

Remark that in the proposed life-cycle FSM a state triggers the execution of 'actions' by other entities. These actions happen when being in a state, while transitions are light weight event based transitions. This forms a difference with the life cycle FSM of Orocos, where the actions, i.e. the execution of configuration etc., happen in between states. The advantages are that 1. the life-cycle FSM can react on errors when executing these actions, 2. a state of the life-cycle FSM can be divided in sub-FSM to coordinate this transition to the level of desired granularity.

5.8 Communication

Communication relates to the **exchange of data** [32, 131]. Different communication mechanisms are possible, for example data flow, events, and service calls.

5.8.1 Modeling

The communication follows the commonly used connector design pattern [12, 149] that decouples dataflow between entities by abstracting the locality of the entities. It enforces a communication protocol. Figure 5.1 shows the different communication mechanism within a Composite `Functional Entity`. `Functional Entities` exchange *data flow*. `Monitors` monitor this data flow and communicate events. `Coordinators` exchange *events* with all entities of a Composite Functional Entity, as well as with the `Coordinators` of a higher and lower level of composition. `Schedulers` interact with `Functional Entities`, and possibly the `Schedulers` of the higher and lower composition levels by *service calls* or events. In addition they exchange events with the `Coordinator`.

5.8.2 Implementation

The reference implementation uses mainly the Orocos port infrastructure, with connections between them. Orocos provides lock free, thread-safe communication and integrates with ROS topics or middleware such as CORBA. The `Composer` creates these connections.

An important setting for the communication of events is the buffer of the connection. Since multiple (common) events can occur at any time, and `Coordinators` advance when (time-)triggered, multiple events can accumulate

between two executions of a `Coordinator`. Moreover an entity has multiple event sources. A buffer must be used to avoid the loss of events. An entity has to empty this buffer when reading from the port receiving events.

A major drawback in communication are the many data types available to represent the same content. Moreover, majority of these data types are general and have no specific semantic meaning. In the reference implementation a tag is provided to all entities that communicate data to specify this semantic meaning. This tag specifies the data model and the meta-model, if it exist.

The `Composer` uses these tags, together with model information from for example an iTaSC DSL model, to automatically resolve connections between entities.

5.8.3 Discussion and lessons learned

The Composite `Functional Entity` as boundary of knowledge helps to reduce the number of events communicated throughout the levels of composition. Events of entities other than the `Coordinator` or `Scheduler` can be configured to remain within that boundary.

As mentioned are many data types available to represent the same content, and they mostly lack a semantic specification. A promising approach is standardisation of notations and specific models of these semantics. An example is the work by De Laet et al. [42, 43], to standardise semantics for geometric relations. They also provide software support to enhance common data types for geometry with these semantics. The following workflow shows how geometric relation semantics integrates in the presented approach:

- each Port should get a model of the data it makes available;
- that model should be in a standardized semantic format;
- when the `Composer` is making the interconnection between components, it should check whether the semantic model (and meta-model) of both Ports are the same;
- in case both Ports have different implementations of the model, transformation code could be added automatically (if such code is available in the binaries of the system).

The implications on the overall design are: (i) the Communication and `Composer` activities must be made aware of the semantic models, and (ii) they

must have access to implementations that support the model checking and transformations. These implications are almost trivial, conceptually speaking, but horrendously huge for the design and implementation of code. Currently, the authors are not aware of one single software project that supports even the simplest form of such semantic awareness.

5.9 Conclusions

This chapter introduces, motivates, and illustrates two major "best practices" that resulted from the accumulated experience of dozens of person years of robotic software framework development at the authors' research group. The first "best practice" is that of the 5Cs principle of separation of concerns [91,128]: the **c**ommunication, **c**omputation, **c**oordination, and **c**onfiguration aspects of any software project should be kept fully separated, but ready to be integrated into a **c**omposition architecture. For the latter, we introduce a second "best practice", the Composition Pattern, that has proven to be very helpful as the basic building block in the design of application-specific, complex system architectures. (A third, derived, "best practice" might be the insight that starting a complex system development process with imposing a specific system architecture from the start is a recipe for failure in the long term.)

The chapter illustrates the general best practices by means of the recent intensive refactoring of our *iTaSC* software framework, a generalized constraint-based programming approach [46] (Section 5.3.2), because (i) it was the application in which the authors first encountered the fundamental deficiencies of former design "guidelines", and (ii) task specification, execution and monitoring involves "planning", "sensing", "control", and "world modeling" functionalities, hence it is a primary example of a robotics system. It is also that broad system integration context and challenge that is the major difference between robotics and other software developments for engineering systems.

The reference implementation uses, in itself, two other large-scale software frameworks, *Orocos* [33] and rFSM [91]; all of them are available under open-source licenses, so readers have access to all details about to what extent exactly we have succeeded in realising the documented best practices in the actual code.

Our search for (i) a systematic way of *describing tasks* in *iTaSC*, together with (ii) the *reusability* driver in the *software implementation* of the iTaSC software framework, drove our software development approach strongly towards a *formalization* of our functionalities and software by means of *domain-specific languages* (DSLs); the result in the context of *iTaSC* can be seen from Vanthienen et al. [167].

More concretely, we here enclose a critical discussion of the *lessons learned* in the design and application of the presented "best practices":

- Separation of concerns is a mainstream design driver, but is often used in isolation, i.e. 'separation of concerns hence reusable entities'. We learned that *composition is as important as separation*. This is *the* difference between the 4C's of Radestock et al. [131], and the 5Cs as used in this chapter, which explicitly focuses on (structural) *Composition*.
- We have (mis)led ourselves during more than a decade in believing that *"components"* are the fundamental building blocks for reusability of functionalities in various architectural compositions. Now, the more complex but very structured and motivated *Composition Pattern* of Figure 5.1 has become the first-class citizen in our system design. Components are still necessary building blocks, but they should not be the *fundamental* building blocks anymore. This is a very important difference, since a component that is *designed* to be part of the Composition Pattern will be different from a component that is designed without that context, since the explicit separation of the `Coordinator`, `Composer`, `Scheduler`, `Configurator`, `Communicator`, `Monitor`, and `Functional Entity` improve the different qualities (the "ilities" such as adaptability, reusability, etc.) of the building blocks. The first four entity types "manage" the last two, keeping the component flexible during usage, hence improving their adaptivity and adaptability. Moreover, this separation distinguishes application specifics (for example concrete controller gains, monitored conditions to switch controllers, or the succession of the control algorithms to use). Only by exception, one or more of the various parts of the Composition Pattern are left out in a concrete design.
- The *modeling* of software has become second nature to us, since thinking about which DSL(s) would be needed to let non-software (but domain) experts exploit our software frameworks, has been proven to be a strong driver for more structured coding.
- The emphasis on modeling is only becoming more and more important, the closer robotics moves towards "cognitive" robot systems, because the latter *have* to be able to reason about their own functionalities, structure and behavior. Such reasoning is only possible when formal, symbolic models of those aspects are available, so the DSLs are expected to be disruptive in that area too.
- The Composition Pattern introduces a significant number of "design forces", which take a bit more time to grasp fully than the more simple

5Cs. The advantage however, is that this more elaborate structure results invariably in much smaller configuration files or software libraries, because developers find it a lot easier to define the scope of each particular software development effort.

This chapter focuses on structure. An important complementary research topic, outside the scope of this chapter, are formal verification and validation tools, which check consistency of the different models used in an application.

None of the above-mentioned lessons learned, and neither the 5Cs nor the Composition Pattern, are derived from unshakable "first principles", hence they can, and should, be subject of continuous critical reflections. The higher than usual degree of structure in the presented material should make such refutation a lot easier; but it is this same "easiness" with which human developers can grasp this structure that has led to the maturation of the concepts, and the clarification of the "design forces", to a level that has stood firmly against dozens of new software project developments, as well as refactorings of existing frameworks.

Chapter 6

Force-sensorless Wrench Control*

6.1 Introduction

Wiping a table, screwing a screw in a hole, pushing a cart, or co-manipulate a table with a human are all representative service robotic tasks with an important common capability requirement: the ability of the robot to exert forces and torques on its environment. Common force control schemes require a force measurement or estimate. In the first case a force sensor needs to be included, which adds to the cost of the platform and alters the dynamics of the robot arm; in the latter case a precise dynamic model of the robot and its environment is required. However, the PR2 robot used as example service robot in this chapter, does not have force sensors, nor has it a precise dynamic model available. In addition, many service robotic tasks such as the ones mentioned above, do not require precise force control. For example wiping a table requires *some* force to be applied, whether five or ten Newtons is of little relevance. Moreover, service robots have to simultaneously combine this ability to exert forces with

*This chapter is partially based on Vanthienen, D., De Laet, T., Decré, W., Bruyninckx, H., De Schutter, J. (2012), "Force-Sensorless and Bimanual Human-Robot Comanipulation", *10th IFAC Symposium on Robot Control*, Dubrovnik, Croatia, 5-7 September 2012 (pp. 1-8), and the preliminary work carried out in the master theses of Jesús Pascual Hernández (2012), "Robotic table wiping: force sensorless robot-world interaction using iTaSC" and Steven Robyns (2013), "Ontwikkeling van een krachtcontrolestrategie zonder krachtsensor voor huishoudelijke robotmanipulatietaken."

other task and safety requirements (*constraints*) such as joint-limit and obstacle avoidance constraints, tool- and self-positioning constraints, etc.

To address this need, this chapter introduces and analyses simple force control schemes that fit the resolved velocity iTaSC control scheme, without the need for a force sensor nor a precise dynamic model of the robot and its environment[†]. This integration allows the combination of force control task constraints with other task constraints, using resolved-velocity constrained optimization.

A common but important restriction in many robot platforms is the limited access to the lower-level controller settings (i.e. inner control loops: controllers on individual joints, or subsets of joints). Due to this restriction, as well as the optimal setting of the lower-level controllers with respect to the overall set of tasks to execute, **this chapter assumes that the gains of the lower-level controllers are known but cannot be altered** (*'grey box'*). Moreover, **this chapter assumes that** (i) **the robot joints involved in the control scheme are backdrivable,** (ii) **and the lower-level[‡] joint velocity control loops have only proportional gains.** These assumptions result in joint velocities and hence joint velocity errors that reflect the effects of possible joint-torque disturbances.

This chapter introduces and analyses different control schemes, which are variations of the abstracted overview scheme of Figure 6.1 in the one-dimensional case, and of Figure 6.2 in the multi-dimensional case. These control schemes consist of three parts:

- The part to the left shows the high-level control loop, consisting of
  - a reference adaptation loop, which will adapt the low-level control loop gain, as will be detailed in Section 6.3,
  - a feedforward factor $\boldsymbol{FF}$, transforming a desired torque τ_d or (part of a) wrench $\boldsymbol{w}_{dsel}$ to a desired joint velocity $\dot{\boldsymbol{q}}_{ex}$ or task space velocity $\dot{\boldsymbol{y}}_{ex}$ in the multi-dimensional case,
  - the pseudo-inverse of an augmented Jacobian $\boldsymbol{A}^{\#}$ in the multi-dimensional case, to transform the desired task space velocity to the robot joint space, and
  - other task constraints expressed as desired velocities $\dot{\boldsymbol{y}}^{\circ}_{d,2..i}$.

  The reference adaptation loop and the feedforward factor form together the wrench (or part thereof) control loop.

[†]This is preferred over estimating the forces by measuring the currents in the actuators, because these estimates are disturbed by joint friction forces

[‡]This dissertation refers to the inner loop as the low-level loop, and the outer loop as the high-level loop.

- The part in the middle shows the low-level control and system model, consisting of
  - the system model, which has a desired (vector of) torque(s) $\boldsymbol{u}$ as input and (measured) (vector of) joint velocity/ies $\dot{\boldsymbol{q}}$ as output , and
  - the low-level control loop wich multiplies the (vector of) joint velocity error(s) $\boldsymbol{e}_{\dot{\boldsymbol{q}}}$ with the (diagonal matrix of) proportional control gain(s) $\boldsymbol{K}_v$.
- The part to the right shows the contact model, resulting in the contact torque τ or wrench $\boldsymbol{w}$.

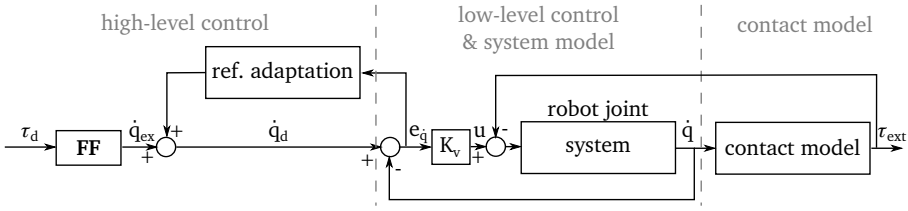


Figure 6.1: Abstracted overview scheme for the **one-dimensional case**.

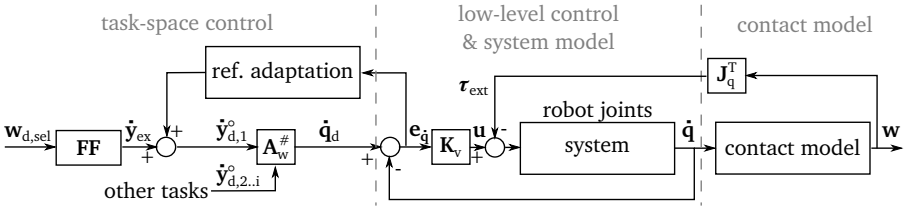


Figure 6.2: Abstracted overview scheme for the **multi-dimensional case**.

Each section details which part of the abstracted overview schemes will be discussed in the section.

This chapter is structured as follows: Section 6.2 states related work and theoretical background. Section 6.3 presents a first, special case of force-sensorless wrench-nulling control used to achieve natural human-robot comanipulation. This wrench-nulling control scheme is analysed for the one-dimensional case, and experimentally validated for the six-dimensional case. Section 6.4 analyses the one-dimensional force/torque control scheme. Section 6.5 analyses and simulates the effect of the reference adaptation loop in a multi-dimensional space. Section 6.6 elaborates the six-dimensional force/torque control scheme. Section 6.7 presents the experimental validation of the proposed

control schemes. Section 6.8 discusses the relation of the presented control schemes with related work. Finally, Section 6.9 summarizes the contributions, states application guidelines, and discusses future work.

Each of the sections provides an introduction and conclusion which formulate the contribution of the section. This allows the reader to follow the reasoning of the chapter, without the need to focus on the details.

6.2 Related work and theoretical background

This section briefly reviews related work in addition to the discussion of Chapter 2. Furthermore, this section gives an overview of some important theoretical principles that will be used in this chapter.

6.2.1 Related work

The majority of robots exhibit coupled, non-linear system dynamics. There are two common approaches to **decouple the joints of the robot** [168]:

- One approach uses feedback linearization, e.g. inverse dynamic model linearization, to decouple and linearize the robot dynamics. This approach uses the inverse dynamic model to transform a desired joint acceleration vector to a desired joint torque vector, using the (estimated) robot dynamic model.
- Another approach relies on high bandwidth (proportional-integral) joint velocity controllers for each joint. This approach largly decouples robot dynamics in case the environment-robot system is sufficiently compliant, and in case the desired motions are relatively slow with respect to the joint velocity controller bandwidth.

The first approach uses an acceleration-based control formulation, also known as *acceleration-resolved control* or *resolved-acceleration control*; the second approach uses a velocity-based control formulation, also known as *velocity-resolved control* or *resolved-velocity control*.

The control schemes presented in this chapter focus on service robot tasks. Service robots such as the PR2 robot used in the experimental validation of this chapter, incorporate some compliance by design. This compliance is one of the measures to ensure safe interaction with humans. **This chapter presents**

resolved-velocity control schemes, since the compliance allows it and in order to avoid the need of (i) an accurate dynamic model of the robot and its environment (ii) and a force-torque sensor.

However, the application of a resolved-velocity control scheme does not imply that no force-torque sensors are needed. The use of a disturbance observer is a **common approach to avoid these force-torque sensors**. Different approaches to estimate the external wrench based on a disturbance observer are presented in literature [59, 84, 151]. These approaches require a precise dynamic model of the robot. Since such a model is unavailable for the PR2, and the application does not require accurate wrench control, **we avoid a disturbance observer**[§].

Furthermore, active wrench control approaches can be divided in two categories [168]: direct and indirect wrench control approaches. *Direct wrench control* approaches achieve wrench control through wrench feedback loop closure, *indirect wrench control* achieves wrench control through motion control, without the need for explicit closure of a wrench feedback loop.

Direct wrench control approaches such as *hybrid (wrench/motion) control* [132] divide the specification of wrench and motion control. If designed well, they show good tracking performance. Indirect approaches such as *impedance or admittance control* [70–72, 85] specify the desired motion and the desired dynamical behavior of the robot (i.e. its *impedance*). They exhibit a trade-off between tracking performance and limiting the contact wrench.

Impedance control uses the expression of the desired dynamical behavior to transform a desired motion (and wrench) to a *desired torque or equivalent joint acceleration*. In contrast, *admittance control* uses the expression of the desired dynamical behavior to transform a desired wrench (and motion) to a *desired motion*.

Section 6.8 discusses the relation of the control schemes presented in this chapter with hybrid control and impedance/admittance control.

The control schemes presented in this chapter fit in the iTaSC approach to constraint-based programming [46], which generalizes hybrid force/motion control to a generalized task space. Section 2.2.2 details constraint-based programming and the iTaSC approach.

[§]However, the reference adaptation loop, presented in Section 6.3, could be seen as a simple estimation scheme, since the feedback value presents a measure for the applied wrench.

6.2.2 Algebra

This section gives an overview of some important concepts from algebra that will be used in this chapter. This dissertation only considers matrices of which the elements are real numbers.

Orthogonal projection matrix

An orthogonal projection matrix $\boldsymbol{P}$ is a matrix that is symmetric (Hermitian) and idempotent. The first implies that $\boldsymbol{P}^T = \boldsymbol{P}$, the latter implies that $\boldsymbol{P}^2 = \boldsymbol{P}$.

The Moore-Penrose pseudo-inverse

This paragraph considers only a limited number of properties of the Moore-Penrose pseudo-inverse, a more in-depth explanation can be found in [17].

Given a matrix $\boldsymbol{A}$ of size $(m \times n)$ with $m < n$, and $\boldsymbol{A}^{\#}$ the Moore-Penrose pseudo-inverse of $\boldsymbol{A}$, following two projection matrices can be defined:

$$\boldsymbol{P} = \boldsymbol{A}\boldsymbol{A}^{\#}, \text{ and} \tag{6.1}$$

$$\boldsymbol{Q} = \boldsymbol{A}^{\#}\boldsymbol{A}. \tag{6.2}$$

$\boldsymbol{P}$ is the orthogonal projection matrix on the range of $\boldsymbol{A}$, and $\boldsymbol{Q}$ is the orthogonal projection matrix on the range of $\boldsymbol{A}^T$. If $\boldsymbol{A}$ is of full row rank, $\boldsymbol{P} = \boldsymbol{I}_{(m\times m)}$. In this equation $\boldsymbol{I}_{(m\times m)}$ denotes the identity matrix of size $(m \times m)$. However, $\boldsymbol{Q} \neq \boldsymbol{I}_{(n\times n)}$, it will be a $(n \times n)$ matrix of rank m.

Given a second matrix $\boldsymbol{B}$ of size $(n \times l)$, the equation

$$(\boldsymbol{A}\boldsymbol{B})^{\#} = \boldsymbol{B}^{\#}\boldsymbol{A}^{\#} \tag{6.3}$$

holds if:

- $\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{I}_{(n\times n)}$, i.e. $\boldsymbol{A}$ has orthonormal columns,
- $\boldsymbol{B}\boldsymbol{B}^T = \boldsymbol{I}_{(n\times n)}$, i.e. $\boldsymbol{B}$ has orthonormal rows, or
- $\boldsymbol{A}$ is of full column rank and $\boldsymbol{B}$ is of full row rank.

Selection matrix

This chapter will make use of selection matrices to select a sub-space of a task space. A selection matrix consists of a matrix with only 0 or 1 elements, with

each column and row having maximal one element equal to 1. The selection matrix $\boldsymbol{S}$ is a semi-orthogonal selection matrix of full column rank, implying the columns are orthonormal vectors and $\boldsymbol{S}$ is semi-unary. Therefore, $\boldsymbol{S}^{\#} = \boldsymbol{S}^{T}$ and $\boldsymbol{S}^{\#}\boldsymbol{S} = \boldsymbol{I}$, however $\boldsymbol{S}\boldsymbol{S}^{\#} \neq \boldsymbol{I}$. For example the matrix to select the x and y directions of a Cartesian task space is a selection matrix of dimensions (6×2), equal to

$$\boldsymbol{S} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}. \tag{6.4}$$

6.3 Special case: one-dimensional reference adaptation loop and force-sensorless wrench nulling

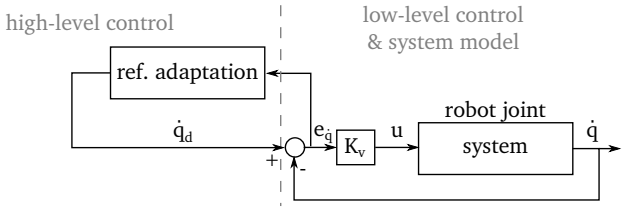


Figure 6.3: Part of the abstracted overview scheme for the one-dimensional case, analyzed in Section 6.3.

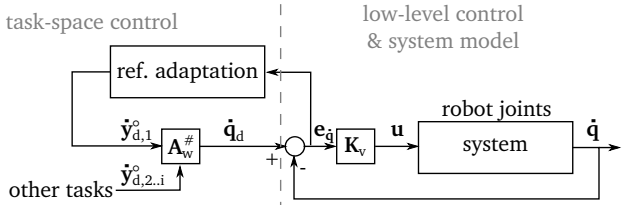


Figure 6.4: Part of the abstracted overview scheme for the multi-dimensional case, experimentally validated in Section 6.3.

An important aspect of human-robot co-manipulation tasks, as elaborated further on in Chapter 7, is the capability of the robot to sense and react on the forces and torques applied by the human on the manipulated object.

More specifically, the robot has to hold the manipulated object, and follow the movement of the human. The human indicates this movement by exerting a wrench on the manipulated object. This section elaborates a force-sensorless wrench nulling control scheme, in which robot tries to eliminate the forces and torques applied on it in a certain task space, effectively resulting in the following behavior required by human-robot co-manipulation tasks. The section first analyzes the one-DOF case, of which the abstract scheme is shown in Figure 6.3, and then experimentally validates the multi-DOF case, of which the abstract scheme is shown in Figure 6.4.

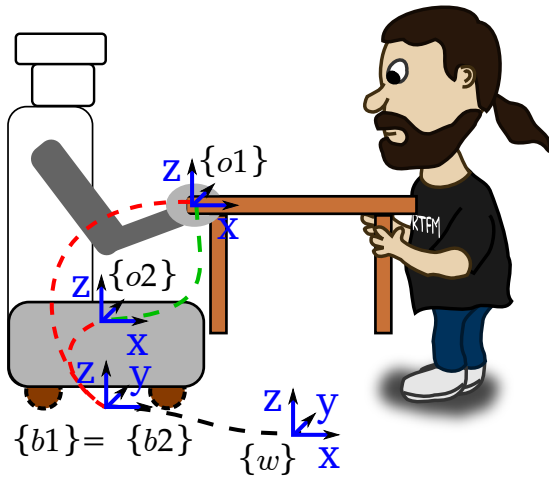


Figure 6.5: The kinematic loop of a wrench-nulling task of human-robot comanipulation, with indication of the world frame $\{w\}$, base frames $\{b_1\}$ and $\{b_2\}$, and object frames $\{o_1\}$ and $\{o_2\}$

Figure 6.5 gives an example **wrench-nulling task** as applied in the human-PR2 co-manipulation application of Chapter 7[¶]. This application constrains a *Cartesian feature space* between the object frame on a gripper $\{o1^n\}$ and an object frame coinciding with the mobile base of the robot $\{o2^n\}$. The first feature frame $\{f1^n\}$ coincides with $\{o1^n\}$, and the second feature frame $\{f2^n\}$ coincides with $\{o2^n\}$. The feature coordinates defining the six-DOF of the resulting *VKC* are all located between $\{f1^n\}$ and $\{f2^n\}$, i.e. $\boldsymbol{\chi_{fI}}^n = 0$ and $\boldsymbol{\chi_{fIII}}^n = 0$, while an intuitive definition of $\boldsymbol{\chi_{fII}}^n$ is obtained by expressing these coordinates in the second feature frame $\{f2^n\}$:

$$\boldsymbol{\chi_{fII}}^n = [x^n, y^n, z^n, \phi^n, \theta^n, \psi^n]^T , \tag{6.5}$$

[¶]Chapter 2 gives an overview of the terminology applied in this section.

where $[x^n, y^n, z^n]^T$ define the $3D$ position coordinates of the gripper with respect to the mobile base of the robot and $[\phi^n, \theta^n, \psi^n]$ is a set of Euler-angles defining the orientation between the gripper and the mobile base of the robot. The output of interest is the wrench $\boldsymbol{y}^n = \left[F_x^n, F_y^n, F_z^n, M_x^n, M_y^n, M_z^n\right]^T$, defined in the *Cartesian feature space* between the object frames on respectively the robot gripper and the mobile base of the robot.

The output equations resulting from the control schemes of all tasks form an optimization problem. A solver resolves the optimization problem, resulting in the desired joint velocities of the low-level velocity controller of the PR2. Hence the wrench-nulling control action operates on this task level and not directly on the low-level velocity controller.

6.3.1 Control scheme and theoretical analysis

To simplify the analysis we first focus on one-DOF, i.e. one rotational robot joint (Figure 6.6). The low-level proportional velocity controller has a gain K_v. The joint motor model includes inertia I_r and damping c_r. If the operator applies an input wrench $\boldsymbol{d}$ to a gripper of the robot, the corresponding robot arm will move, due to its backdrivability. In the one-DOF case, this input wrench reduces to a disturbance torque d, which causes a joint velocity $\dot{q}$ different from the desired joint velocity $\dot{q}_d$, expressed by velocity error $e_{\dot{q}} = \dot{q}_d - \dot{q}$. The relation of this velocity error $e_{\dot{q}}$ with the disturbance torque d can be written in the Laplace domain as:

$$e_{\dot{q}} = \frac{(I_r s + c_r)\dot{q}_d - d}{I_r s + c_r + K_v}, \tag{6.6}$$

with s the Laplace operator.

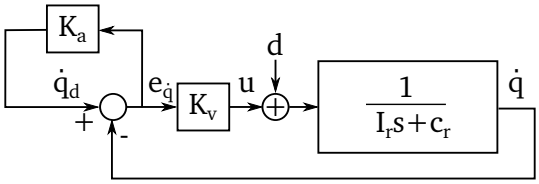


Figure 6.6: One-DOF wrench-nulling control scheme.

By applying a control law K_a (at task level[‖]), a new desired velocity $\dot{q}_d$ is calculated out of the velocity error $e_{\dot{q}}$, described by

$$\dot{q}_d = K_a e_{\dot{q}}. \tag{6.7}$$

Therefore, we will name this high-level control loop the **reference adaptation loop** and its control gain K_a the **reference adaptation factor**.

K_a is chosen in order to reinforce the velocity error caused by the disturbance torque d, such that the operator senses less resistance of the robot mechanism, i.e. the robot assists the operator. The transfer function T_e from the disturbance torque d (the input) to the velocity error $e_{\dot{q}}$ is described by,

$$T_e = \frac{e_{\dot{q}}}{d} = \frac{1}{(K_a - 1)(I_r s + c_r) - K_v}. \tag{6.8}$$

Accordingly, the transfer function T_v from the disturbance torque d to the joint velocity $\dot{q}$ can be written as following first-order system

$$T_v = \frac{\dot{q}}{d} = \frac{1}{I_r s + c_r + \frac{K_v}{1-K_a}} = \frac{A}{s\tau + 1}. \tag{6.9}$$

The following paragraphs analyse the pole p, time constant τ, and gain A of the latter transfer function T_v.

The pole p of T_v equals $(\frac{K_v}{K_a - 1} - c_r)\frac{1}{I_r}$, which is a real number. The pole is negative, hence the system is stable for any value $K_a < 1$ or $K_a > \frac{K_v + c_r}{c_r}$. The latter limit of stability,

$$K_a = \frac{K_v + c_r}{c_r} = \frac{K_v}{c_r} + 1, \tag{6.10}$$

describes the situation in which all damping is compensated by the reference adaptation loop.

The time constant τ of the system characterized by T_v is given by

$$\tau = \frac{I_r}{c_r + \frac{K_v}{1-K_a}}, \tag{6.11}$$

and the gain A by

$$A = \frac{1}{c_r + \frac{K_v}{1-K_a}}. \tag{6.12}$$

[‖]In practice, K_a is applied after transformation of the velocity error from the joint space to the task space using the Jacobian of the robot arm. The desired velocity results from the iTaSC solver and involves a transformation back to joint space. In case of a non-redundant robot and a scalar K_a (or equivalently a diagonal matrix with equal diagonal elements), the transformation from the joint space to the task space and back cancel out.

In effect K_a alters the low-level control gain K_v to

$$K_b = \frac{K_v}{1 - K_a}, \tag{6.13}$$

further denoted the **equivalent low-level control gain**.

Figure 6.7 shows the time constant τ and gain A in function of K_a. Both differ only by a factor equal to the inertia I_r. Consequently, the choice of K_a is a trade-off between a higher gain and a slower response versus a lower gain and a faster response.

A closer look on equation (6.11) shows that the time constant of the open-loop system

$$\tau_{OL} = \frac{I_r}{c_r}, \tag{6.14}$$

i.e. when $K_v = 0$, is the asymptote for τ, i.e. when K_a approaches infinity. Similar conclusions hold for the gain

$$A_{OL} = \frac{1}{c_r}. \tag{6.15}$$

The magenta dash-dotted line in Figure 6.7 shows the open-loop time constant τ_{OL} and gain A_{OL}. The numerical values in the figure apply for the elbow joint of the left arm of the PR2 robot.

Furthermore, the time constant and gain of the low-level velocity loop can be found at $K_a = 0$, i.e.

$$\tau_{VL} = \frac{I_r}{c_r + K_v}, \tag{6.16}$$

and

$$A_{VL} = \frac{1}{c_r + K_v}, \tag{6.17}$$

respectively. The black dash-dotted line in Figure 6.7 shows the low-level velocity loop time constant τ_{VL} and gain A_{VL}.

We choose a gain A higher than the gain of the low-level velocity loop A_{VL}, as a result increasing the sensitivity to the input wrench and hence creating robot assistance. We choose the time constant τ smaller than the open-loop time constant τ_{OL}, hence a faster response than open-loop. Therefore, the area of interest lies between the dash-dotted lines on Figure 6.7. Remark that positive values for K_a do not satisfy the desired behavior, since they have a slower response time than the open-loop system.

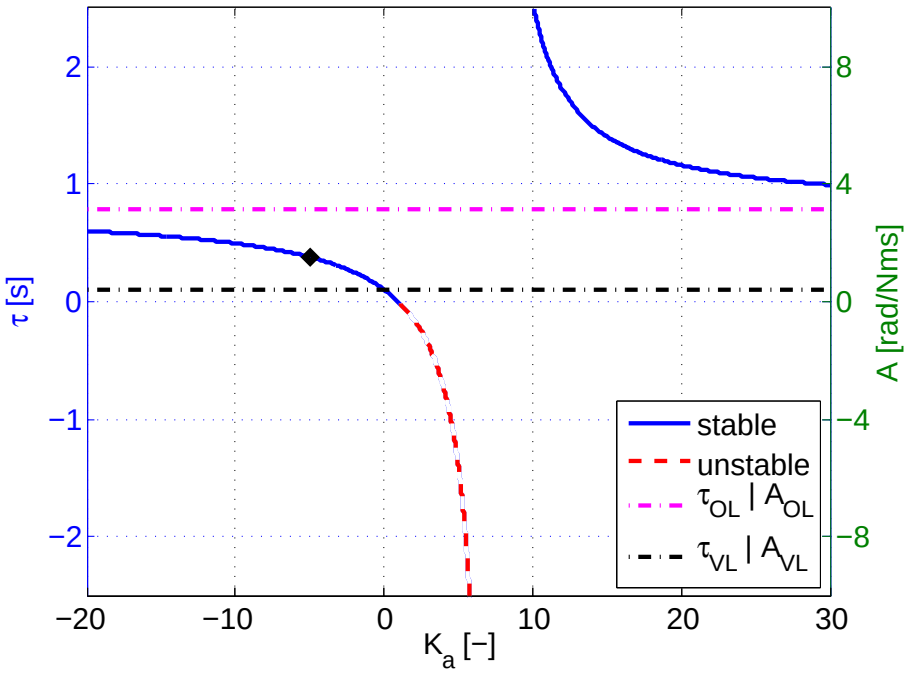


Figure 6.7: Time constant τ and gain A in function of control factor K_a. The axis to the left indicates the value of the time constant, the axis to the right indicates the value of the gain. The following parameters apply for the joint: damping $c_r = 0.32\frac{Nms}{rad}$, inertia $I_r = 0.25\frac{Nms^2}{rad}$, and velocity controller gain $K_v = 2\frac{Nms}{rad}$. The system is stable for any value $K_a < 1$ or $K_a > \frac{K_v + c_r}{c_r}$, as indicated by the blue line. The red dashed line indicates the unstable part with negative gain values. The part of the curve between the low-level velocity loop gain A_{VL} and the open loop time constant τ_{OL}, marked by the black dash-dotted line and magenta dash-dotted line respectively, indicates the area of interest. The black dot marks the time constant and amplitude for the chosen $K_a = -5$.

6.3.2 Experimental validation

This section provides quantitative results for force-sensorless wrench nulling, using a set-up that forms part of the human-robot co-manipulation application detailed in Chapter 7. For repeatability of the experiments, the force applied by the human is replaced by a constant pulling force. A mass of 1.5kg, connected through a pulley system to the PR2 robot's left gripper delivers this pulling force. When the mass is released, the weight of the mass pulls the gripper with a constant force in a direction approximately aligned with the x-direction. In the initial state, i.e. before applying the force, the robot arm joints are far from their limits, with the elbow pointing downwards. Figure 7.9 shows the experimental setup.

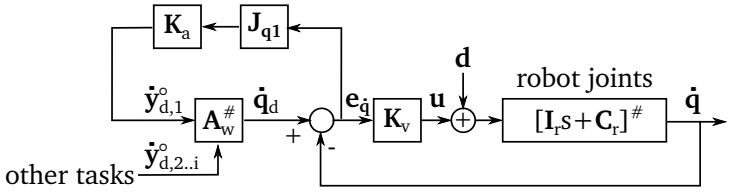


Figure 6.8: Multi-DOF wrench-nulling control scheme.

The experiments extend the control scheme to a multi-dimensional space, as shown in Figure 6.8. In this control scheme for the multi-dimensional case, the reference adaptation loop acts in the wrench-nulling task's Cartesian feature space.** The desired task space velocity results from the projection of the joint velocity errors $\boldsymbol{e}_{\dot{q}}$ in the task space, multiplied by the reference adaptation factor K_a. The reference adaptation loop only uses the joint velocity errors of the *left arm* of the robot to calculate the desired task space velocity since these joints are backdrivable. However, the projection of the desired task space velocity back to the joint space of the robot projects this velocity on the *whole robot joint space* involved in the kinematic loop of the wrench-nulling task. In the presented example, this includes the robot's mobile base and spine next to the left arm. It is as if the robot's left arm serves as a 'force sensor' input to the reference adaptation loop. Sections 6.5 analyses in detail the effect of the multi-dimensional reference adaptation loop.

Equation (6.8) defines the relation of the error $e_{\dot{q}}$ with the input torque d for one-DOF. A similar relation holds for a direction of the task space, assuming that the transformations between the joint space and the task space cancel out. In case of a redundant robot but scalar K_a this includes a projection in the

**Further on the wrench-(nulling) task's Cartesian feature space is simply referred to as the task or task space.

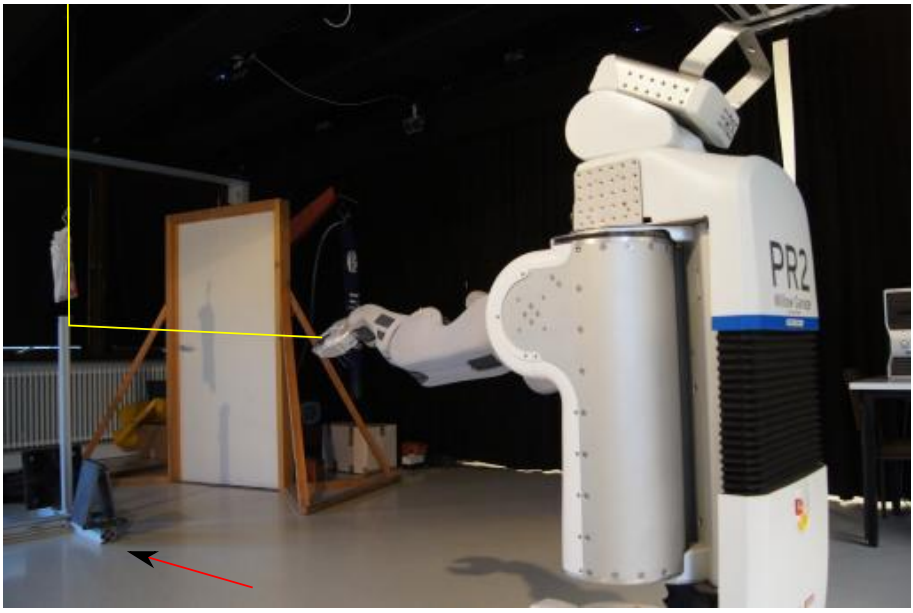


Figure 6.9: Picture of the experimental setup. The cable (yellow) is enhanced for visibility. The cable and pulley system connects the PR2 robot's hand to the white bag with the weight, shown at the left. The the red arrow indicates the x-direction.

column space of the Jacobian $\boldsymbol{J_q}$. Applying a constant force in the x-direction to one of the robot hands as in the previous experiment, causes a velocity error $e_{\dot{x}}$ in this direction, and the movement of the robot in this x-direction (following behavior).

A control factor $K_a = -5$ is chosen for all DOF of the task space in this experiment. Figure 6.10 shows the velocity error $e_{\dot{x}}$ over time, after applying this force. The pitch θ^{ee}_{base} of the end effector's orientation with respect to the robot's mobile base illustrates the transition phase, just after the mass has been released, when the pulled arm makes an up and down movement. This transient is due to the dynamics of each individual joint, as described by equation (6.8), and the complex geometric interaction between the joints, i.e. our simplified assumptions are not satisfied during transients. After this movement damps out, $e_{\dot{x}}$ reaches a constant value. This error is proportional to the applied force, with a negative factor of proportionality, as stated by (6.8). Hence, applying a constant wrench results in a constant assistance, i.e. a constant velocity in the x-direction, after transition behavior.

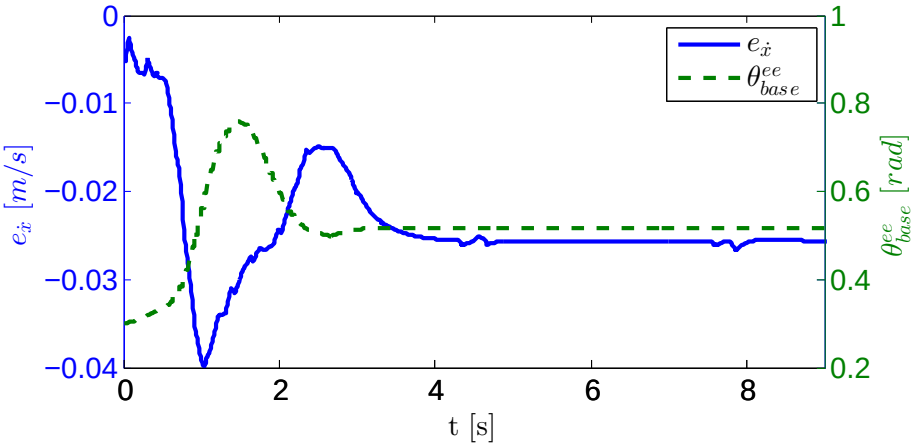


Figure 6.10: The velocity error in the x-direction $e_{\dot{x}}$ and pitch θ^{ee}_{base} of the end effector with respect to the robot's mobile base, in a full blue line and a green dashed line respectively.

6.3.3 Discussion and conclusions

This section described a special case of force-sensorless force or wrench control, i.e. wrench nulling. The wrench nulling control scheme adds a simple (joint- or Cartesian-) task space control loop to the resolved-velocity iTaSC control scheme. This control loop, further denoted the **reference adaptation loop**, effectively adapts the low-level control gains in the selected direction of the task space. In case of wrench-nulling in human-robot co-manipulation, the low-level gain is adapted to reduce the damping felt by the human, providing following behavior by amplification of the human-applied wrench. Experimental validation of the multi-DOF wrench-nulling shows that the robot follows an applied force after transients have damped out, effectively nulling the applied force. A video, made available online [164][††], shows the performance of the wrench-nulling control scheme as part of the human-robot co-manipulation demo detailed in Chapter 7. The wrench-nulling control scheme enables direct human-robot interaction without the use of a force sensor. Further sections will detail general force-sensorless wrench control, and the effect of the reference adaptation loop in multi-DOF.

[††]Appendices C.1 and C.3 list and detail the videos related to this chapter.

6.4 One-dimensional force-sensorless force/torque control

This section analyses the one-DOF case of the force-sensorless wrench control scheme that extends the reference adaptation loop introduced in the previous section with **an offset desired velocity $\dot{q}_{ex}$, different from zero**. The offset should result in a desired wrench, or part thereof, on the environment. This section analyses the control scheme, (i) first in free space, i.e. the approach towards the contact situation when no contact between the robot end-effector and the environment is established yet, (ii) then in contact with the environment. Figure 6.11 shows the abstracted control scheme for the considered situations.

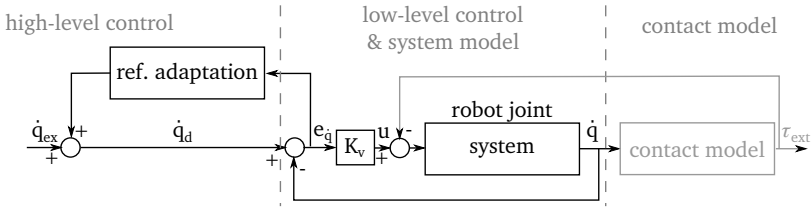


Figure 6.11: Part of the abstracted overview scheme for the one-dimensional force/torque control case, discussed in Section 6.4. The feedforward factor FF is left out (or assumed equal to one), the contact model is left out in the free space case.

6.4.1 Free space

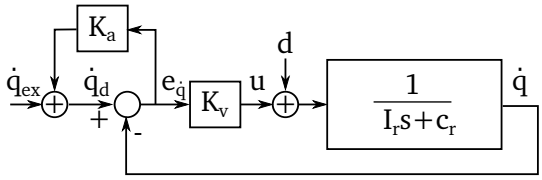


Figure 6.12: one-DOF torque control scheme in free space.

Adding an offset term to the one-DOF control scheme of Figure 6.6, results in the free space control scheme shown in Figure 6.12. In contrast to the special case of Section 6.3, the control scheme should not null an external torque (the disturbance d), but should result in the application of a desired force or torque once in contact with the environment. Hence the transfer function of interest

relates the output velocity $\dot{q}$ with the input offset term $\dot{q}_{ex}$, expressed as

$$T_{ex} = \frac{\dot{q}}{\dot{q}_{ex}} = \frac{\frac{K_v}{1-K_a}}{I_r s + c_r + \frac{K_v}{1-K_a}} = \frac{A_{ex}}{\tau_{ex} s + 1}. \tag{6.18}$$

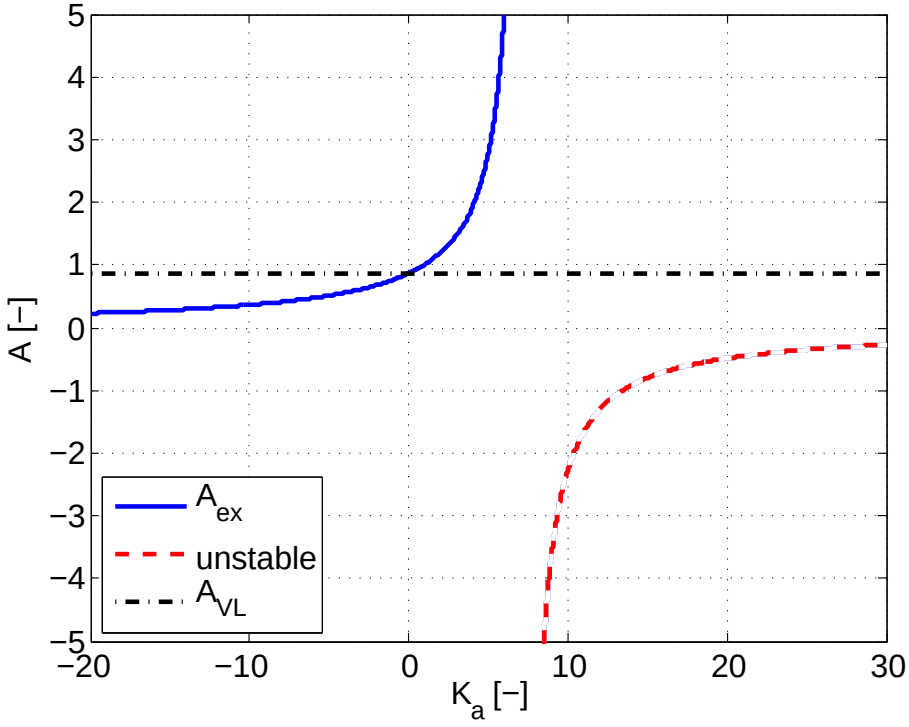


Figure 6.13: Gain A_{ex} in function of control factor K_a for the one-DOF free space system. The red dashed line indicates values of K_a with a negative gain. The black dash-dotted line indicates the low-level velocity loop gain A_{VL}. The vertical assymptote is situated at $K_a = \frac{K_v + c_r}{c_r}$.

The gain and time constant of this first-order system can be written as

$$A_{ex} = \frac{\frac{K_v}{1-K_a}}{c_r + \frac{K_v}{1-K_a}} \tag{6.19}$$

and

$$\tau_{ex} = \frac{I_r}{c_r + \frac{K_v}{1-K_a}}, \tag{6.20}$$

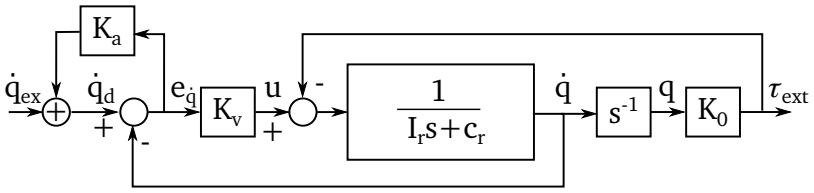


Figure 6.14: one-DOF torque control scheme in contact.

respectively. Remark that K_a alters the low-level control gain K_v to $K_b = \frac{K_v}{1-K_a}$, as in the wrench-nulling case.

Figure 6.13 shows the gain A_{ex} in function of K_a. The gain A_{ex} is positive for any value $K_a < \frac{K_v+c_r}{c_r}$. The time constant τ_{ex} in function of K_a is the same as the one-DOF torque nulling case shown in Figure 6.7. Hence, the time constant τ_{ex} is positive for any value $K_a < 1$ or $K_a > \frac{K_v+c_r}{c_r}$, and the asymptote of K_a towards infinity is the open-loop time constant τ_{OL}. The numerical values in the figures apply for the same robot joint analysed in Section 6.3, i.e. the elbow joint of the left arm of the PR2 robot. The system is stable for positive time constants and positive gain, hence limiting the choice of K_a to $]-\infty, 1\,[$.

Remark that the offset term is a velocity offset, hence **in free space the robot approaches with constant velocity**.

6.4.2 Contact

This section analyses the situation where the robot makes contact with the environment. Therefore, the control scheme of Figure 6.12 is extended with a contact model, as shown in Figure 6.14. **The model regards the environment as a (stiff) spring** characterized by constant K_0.

This second order-system is characterized by following transfer function

$$T_c = \frac{\tau_{ex}}{\dot{q}_{ex}} = \frac{K_v}{1-K_a} \frac{K_0}{I_r s^2 + (c_r + \frac{K_v}{1-K_a})s + K_0} = \frac{A_c}{\frac{s^2}{\omega_c^2} + 2\zeta_c \frac{s}{\omega_c} + 1}. \tag{6.21}$$

The system gain A_c, natural frequency ω_c, and damping ratio ζ_c are, respectively,

$$A_c = \frac{K_v}{1-K_a}, \tag{6.22}$$

$$\omega_c = \sqrt{\frac{K_0}{I_r}}, \text{ and} \tag{6.23}$$

$$\zeta_c = \frac{c_r + \frac{K_v}{1-K_a}}{2\sqrt{I_r K_0}}. \tag{6.24}$$

Remark that the low-level velocity loop gain K_v amplifies the joint velocity error $e_{\dot{q}}$ to the applied robot joint torque u. In steady-state, the applied torque forms an equilibrium with the torque applied on the environment.[‡‡] Therefore, $e_{\dot{q}}$ **forms a measure for the applied torque** and is used in the reference adaptation loop.

The gain of the system A_c equals the equivalent low-level control gain K_b introduced in Section 6.3. The damping ratio is altered by adding the equivalent low-level control gain to the robot joint damping c_r. Figures 6.15 and 6.16 show the damping ratio ζ_c and gain A_c in function of K_a. The numerical values in the figures apply for the same joint as previous sections. The system is stable for positive damping and positive gain, hence limiting the choice of K_a to the same range as the first order system, i.e. $]-\infty, 1[$, as indicated on the figures. The example system of the left elbow joint of the PR2 robot is critically damped at $K_a = 0.873$.

In order to apply a (constant) desired force or torque, this force or torque needs to be transformed to the velocity offset term $\dot{q}_{ex}$. The needed transformation, denoted *FF*, is the inverse of the transfer function T_c. Considering only steady-state, i.e. when the Laplace operator s equals zero, *FF* is the inverse of the equivalent low-level control gain ${K_b}^{-1} = \frac{1-K_a}{K_v}$, hence

$$\dot{q}_{ex} = FF\tau_d = {K_b}^{-1}\tau_d = \frac{1-K_a}{K_v}\tau_d. \tag{6.25}$$

The system is in steady-state when the feedback joint velocity $\dot{q}$ is zero, and the reaction torque τ_{ex} is in equilibrium with the input torque u. Hence it can be intuitively seen that when the feedforward transformation *FF* cancels the equivalent low-level control gain K_b, the input torque u equals the desired force or torque. As a result, **including the feedforward factor *FF* in the system will make the overall system gain equal to one and independent of** K_a.

6.4.3 Discussion and conclusions

This section introduced a simple but effective control scheme consisting of a offset term $\dot{q}_{ex}$ and the reference adaptation loop characterized by the parameter

[‡‡] Neglecting unmodeled disturbances such as Coulomb friction.

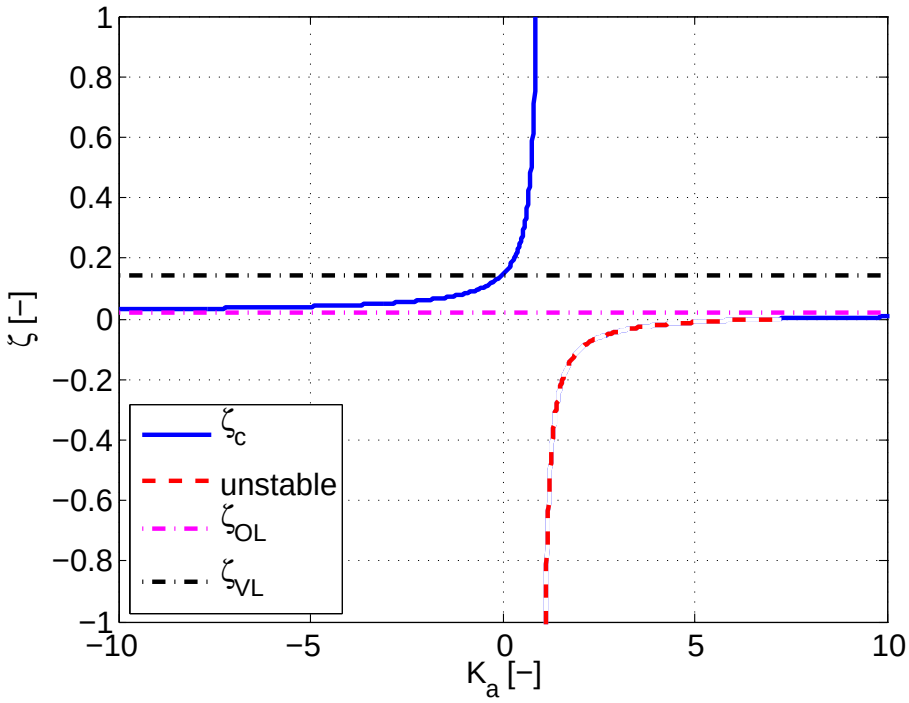


Figure 6.15: Damping ratio ζ_c in function of control factor K_a for the one-DOF system in contact with the environment. The red dashed line indicates the unstable part with negative damping. The black dash-dotted line indicates the low-level velocity loop ($K_a = 0$) damping ratio ζ_{VL}, and the magenta dash-dotted line indicates the open loop damping ratio ζ_{OL}. The latter being the assymptote of ζ_c for large values of K_a.

K_a, operating on a low-level velocity control loop. This low-level velocity loop is considered a 'grey box'.

Two situations are analyzed:

- the system in free space, when the robot approaches the contact situation, but does not make contact with the environment yet, with input the offset $\dot{q}_{ex}$ and output the joint velocity $\dot{q}$, and
- the system in contact with the environment, with input the offset $\dot{q}_{ex}$ and output the torque τ_{ext}.

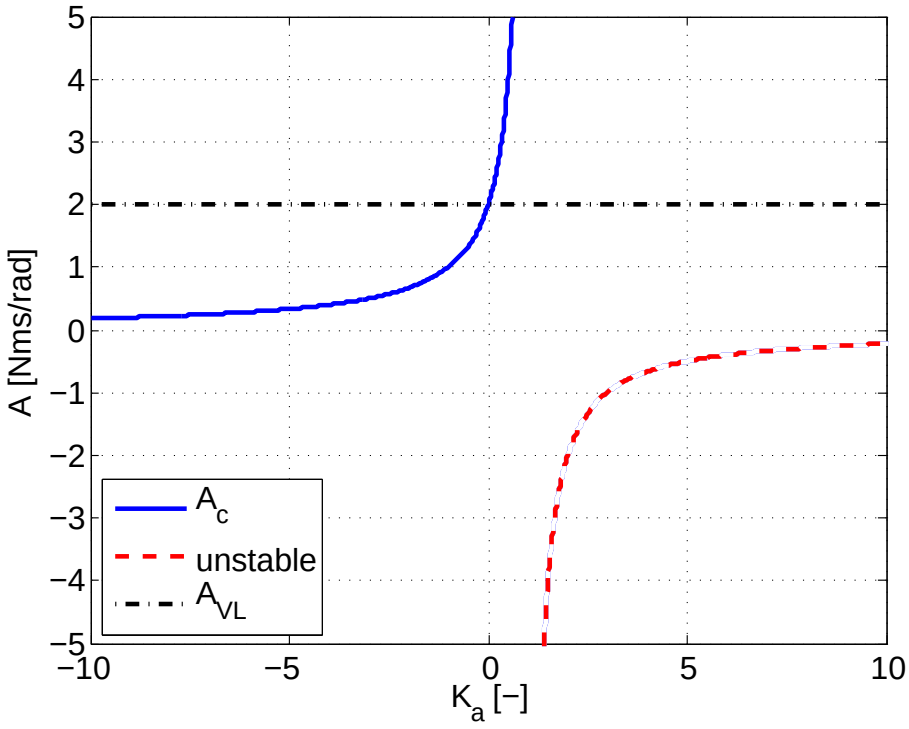


Figure 6.16: Gain A_c in function of control factor K_a for the one-DOF system in contact with the environment. The red dashed line indicates the unstable part with negative gain. The black dash-dotted line indicates the low-level velocity loop ($K_a = 0$) gain A_{VL}.

The former is a first-order system, characterized by the time constant τ_{ex} and (input-output) gain A_{ex}, the latter is a second-order system, characterized by (input-output) gain A_c, natural frequency ω_c, and damping ratio ζ_c.

The reference adaptation constant K_a allows to tune the dynamic behavior of the system by making a trade-off between the desired damping when making contact with the environment, and the time constant τ_{ex} and gain A_{ex} of the free space approach towards this contact situation. The choice of K_a should consider the stability limits and limits on the control input towards the system. The analysis of the system in free space and in contact shows that the value of K_a should be in the $]-\infty, 1[$ range for stability. Within this range, a bigger value of K_a leads to (i) a smaller time constant τ_{ex} and more gain A_{ex} (larger amplification) in free space, (ii) or more damping ζ_c and gain A_c (larger amplification) in contact. The transformation from a desired torque or force to the offset term

$\dot{q}_{ex}$ adds a factor to the system which makes the gain of the system in contact equal to one, hence of no importance in steady-state.

A spring models the contact with the environment, which value influences the natural frequency and damping ratio when the system makes contact. However, it does not influence the steady-state behavior: the feedforward transformation FF and the transfer function of the system in contact T_c are independent of the magnitude of the spring constant K_0. Further, the model does not include disturbances such as friction, which will decrease performance, but will not decrease stability. The main disturbance of concern in the contact situation is Coulomb (static) friction which will decrease the delivered force or torque. In case of a very low desired force or torque the control scheme will be even unable to move the robot joint.

The presented approach shows a number of *advantages with respect to alternative approaches*: (i) The approach does not require an (expensive) force sensor. (ii) The same control scheme can be applied in both free space and contact with the environment for appropriate values of K_a. Hence there is no need for exact contact detection. (iii) The time when and position where the robot makes contact with the environment does not need to be known in order to make a stable contact. (iv) When an appropriate value for K_a is chosen, the system will be stable and behave in a safe and predictable manner since there is no integral control action. The robot joint will move with a constant speed in the direction in which to apply the force or torque, until contact is made. Once contact is made, a force or torque will build up proportional with the offset term.

6.5 Multi-dimensional reference adaptation loop

Figure 6.17: Part of the abstracted overview scheme focusing on the multi-dimensional reference adaptation, discussed in Section 6.5. In contrast to the other multi-dimensional cases of this chapter, this scheme considers a planar robot with a three dimensional task space, and a four dimensional robot.

Previous section analysed a one-DOF force-sensorless force/torque control scheme in free space and contact. This section regards the multi-dimensional control scheme, where the reference adaptation loop and the feed forward term apply to a task space of lower dimensions than the velocity-controlled system. We consider only the free-space motion, in order to study the effect of the multi-dimensional reference adaptation loop, as shown in Figure 6.17.

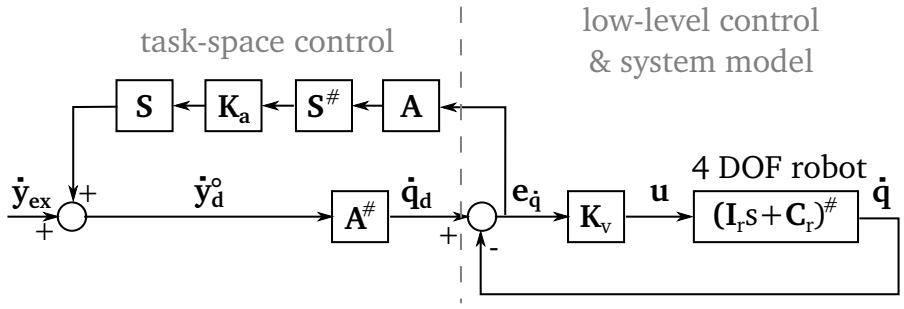


Figure 6.18: Joint-velocity controller with Cartesian task-space reference adaptation loop

This section does consider only under- but not over-constrained task spaces. Therefore, the task space is of equal or lower dimension than the joint space, and the augmented Jacobian $\boldsymbol{A}$ is of full row rank.* Moreover, the reference adaptation loop will include a projection on the column space of the augmented Jacobian $\boldsymbol{A}$. As a consequence the equation for the equivalent low-level control gain $K_b = \frac{K_v}{1-K_a}$ does not hold for the multi-dimensional case, even when all task space directions have the same reference adaptation factor K_a.

To provide greater insight in these effects, this section analyses the example case of a planar serial robot with four rotational joints. Its task space consists of a Cartesian plane described by two position coordinates x and y, and the orientation θ. In the multi-dimensional control scheme, the reference adaptation loop factor $\boldsymbol{K}_a$ is a diagonal matrix of size $(n_c \times n_c)$, with the diagonal elements not equal to 1. The example will consider the more restrictive case where all diagonal elements of $\boldsymbol{K}_a$ are the same, hence $\boldsymbol{K}_a = K_a \boldsymbol{I}_{n_c}$. Since $\boldsymbol{I}_{n_c}$ is the identity operator of a matrix product and to simplify the notation, $\boldsymbol{K}_a$ will be replaced by K_a where possible.

6.5.1 Control scheme

Figure 6.18 shows the control scheme with at the left side the task space control, and the right side the velocity-controlled system. Every joint of the system is controlled separately, hence the proportional gain matrix $\boldsymbol{K}_v$, the inertia matrix $\boldsymbol{I}_r$, and the damping matrix $\boldsymbol{C}_r$ are all diagonal matrices. The augmented Jacobian matrix $\boldsymbol{A}$ transforms the joint velocity error $\boldsymbol{e}_{\dot{\boldsymbol{q}}}$ to the task space; the

*$\boldsymbol{A}$ is of size $(n_c \times n_q)$ The symbol n_c defines the number of task constraints, symbol n_q defines the number of controllable robot joints.

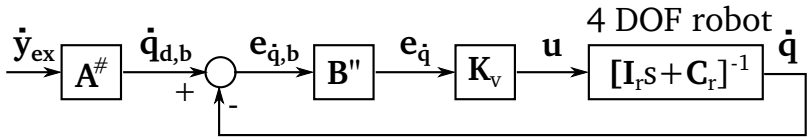


Figure 6.19: Equivalent scheme for the joint-velocity controller with task-space reference adaptation loop

pseudo-inverse of the augmented Jacobian matrix $\boldsymbol{A}^{\#}$ transforms the desired task space velocity $\dot{\boldsymbol{y}}_d^{\circ}$ to the robot joint space. Since the studied example consists of a single Cartesian task space, the augmented Jacobian $\boldsymbol{A}$ equals the Jacobian $\boldsymbol{J}_{\boldsymbol{q}}$. **The selection matrix**[†] $\boldsymbol{S}^{\#}$ **of dimension** $(n_{c,sel} \times 3)$ **selects the directions of the task space on which the reference adaptation will take effect; selection matrix** $\boldsymbol{S}$ **expands the reference adaptation results to the full task space.**

As outlined by Figure 6.18, the joint velocity error can be written as

$$\boldsymbol{e}_{\dot{\boldsymbol{q}}} = \boldsymbol{A}^{\#}[\dot{\boldsymbol{y}}_{ex} + \boldsymbol{S}\boldsymbol{K}_a\boldsymbol{S}^{\#}\boldsymbol{A}\boldsymbol{e}_{\dot{\boldsymbol{q}}}] - \dot{\boldsymbol{q}}. \tag{6.26}$$

Restructuring equation 6.26 and introducing $\boldsymbol{B}'' = \boldsymbol{I} - \boldsymbol{A}^{\#}\boldsymbol{K}_a\boldsymbol{S}^{\#}\boldsymbol{A}$ (full rank if $\boldsymbol{K}_a \neq \boldsymbol{I}$)[‡] results in

$$\boldsymbol{e}_{\dot{\boldsymbol{q}}} = [\boldsymbol{I} - \boldsymbol{A}^{\#}\boldsymbol{S}\boldsymbol{K}_a\boldsymbol{S}^{\#}\boldsymbol{A}]^{-1}[\boldsymbol{A}^{\#}\dot{\boldsymbol{y}}_{ex} - \dot{\boldsymbol{q}}] = \boldsymbol{B}''^{-1}[\boldsymbol{A}^{\#}\dot{\boldsymbol{y}}_{ex} - \dot{\boldsymbol{q}}]. \tag{6.27}$$

$\boldsymbol{B}''$ represents the effect of the reference adaptation loop on $\boldsymbol{K}_v$, resulting in the equivalent gain $\boldsymbol{K}_b = \boldsymbol{K}_v\boldsymbol{B}''^{-1}$, as represented in the equivalent scheme shown in Figure 6.19. Remark that the joint velocity error in the *original scheme* $\boldsymbol{e}_{\dot{\boldsymbol{q}}}$ is actually a value between the two gains $\boldsymbol{B}''$ and $\boldsymbol{K}_v$ in the equivalent scheme. The joint velocity error in the *equivalent scheme* $\boldsymbol{e}_{\dot{\boldsymbol{q}},\boldsymbol{b}}$ can be expressed as

$$\boldsymbol{e}_{\dot{\boldsymbol{q}},\boldsymbol{b}} = \boldsymbol{A}^{\#}\dot{\boldsymbol{y}}_{ex} - \dot{\boldsymbol{q}} = \boldsymbol{B}''\boldsymbol{e}_{\dot{\boldsymbol{q}}}. \tag{6.28}$$

Transforming this error in the task space and refactoring the result, amounts to

$$\boldsymbol{A}\boldsymbol{A}^{\#}\dot{\boldsymbol{y}}_{ex} - \boldsymbol{A}\dot{\boldsymbol{q}} = \boldsymbol{A}\boldsymbol{B}''\boldsymbol{e}_{\dot{\boldsymbol{q}}}, \tag{6.29}$$

$$\dot{\boldsymbol{y}}_{ex} - \boldsymbol{A}\dot{\boldsymbol{q}} = [\boldsymbol{A} - \boldsymbol{S}\boldsymbol{K}_a\boldsymbol{S}^{\#}\boldsymbol{A}]\boldsymbol{e}_{\dot{\boldsymbol{q}}}, \tag{6.30}$$

$$\dot{\boldsymbol{y}}_{ex} - \boldsymbol{A}\dot{\boldsymbol{q}} = [\boldsymbol{I} - \boldsymbol{S}\boldsymbol{K}_a\boldsymbol{S}^{\#}]\boldsymbol{A}\boldsymbol{e}_{\dot{\boldsymbol{q}}}. \tag{6.31}$$

[†]Section 6.2.2 defines and elaborates on selection matrices.

[‡]Appendix A.1 gives a proof for the more general case. Remark also that if $\boldsymbol{K}_a = \boldsymbol{I}$ and $\boldsymbol{S} = \boldsymbol{I}$, $\boldsymbol{B}''$ is the null space projector of $\boldsymbol{A}$.

Solving for the transformed joint velocity error in the original scheme results in

$$\boldsymbol{A}\boldsymbol{e}_{\dot{\boldsymbol{q}}} = [\boldsymbol{I} - \boldsymbol{S}\boldsymbol{K}_a\boldsymbol{S}^{\#}]^{-1}[\dot{\boldsymbol{y}}_{ex} - \boldsymbol{A}\dot{\boldsymbol{q}}], \tag{6.32}$$

showing **a comparable, albeit multi-dimensional effect in the task space as in the one-DOF case**, where $K_b = \frac{K_v}{1-K_a}$ (equation (6.13)).

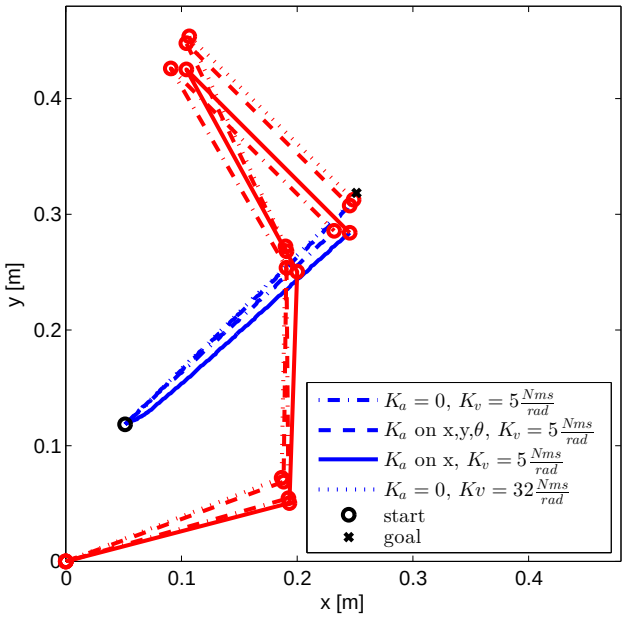


Figure 6.20: Blue lines show paths followed by the robot end effector for different gain settings. Each setting has a different line style, as indicated in the legend of the figure. The red lines indicate the segments of the robot, in the configuration at the end of each of abovementioned paths, i.e. after 2s. The red cirlces indicate the related robot joints. The black circle indicates the start position, the black cross indicates the end position when the robot would track the step input velocity perfectly (integration of the applied velocity over 2s). $K_a = 0.7$ unless explicitly indicated as zero.

6.5.2 Simulations

This section simulates the control scheme for the example case of a four DOF planar serial robot. Figure 6.20 shows four different configurations of this robot

in red. It compares four situations: (i) the reference case of an unadapted velocity control loop, i.e. $K_a = 0$ and proportional gain $K_v = 5\frac{Nms}{rad}$ for each robot joint; (ii) the case where the reference adaptation only acts on one task space direction, i.e. $\boldsymbol{S} = [1\ 0\ 0]^T$ selecting the x direction; (iii) the case where the reference adaptation acts on all task space directions, i.e. $\boldsymbol{S} = \boldsymbol{I}_3$; and (iv) the case of an unadapted velocity control loop with a higher gain $K_v = 32\frac{Nms}{rad} \neq \frac{K_v}{1-K_a}$ for each robot joint. The curves on the figures of this section are in dash-dotted line for the first case, dashed line for the second case, full line for the third case, and dotted line for the fourth case.

The simulated robot consists of four equal robot segments of length $0.2m$, and four equal rotational robot joints. Each of the joints has an inertia $I_r = 1.25kgm^2$ and damping $c_r = 0.32\frac{Nms}{rad}$, and is controlled at $100Hz$ by a velocity controller with a proportional gain K_v. As a result the $\boldsymbol{I}_r$, $\boldsymbol{C}_r$, and $\boldsymbol{K}_v$ matrices are diagonal. The reference adaptation loop constant K_a equals 0.7 for all selected directions. The initial configuration of the robot equals $[0.1\ 2\ 0.7\ 2.7]rad$.

Figure 6.20 shows the trajectories traversed by the robot end effector for the four different cases, when an input of $\dot{\boldsymbol{y}}_{ex} = [0.1\frac{m}{s}\ 0.1\frac{m}{s}\ 0\frac{rad}{s}]$ is applied to the system. The reference case with proportional gain $K_v = 5\frac{Nms}{rad}$ and no reference adaptation has the worst performance of the curves. This performance can be expected for a proportionally controlled first order system with the lowest gain, hence largest steady-state offset. The third case with reference adaptation on all task space directions has a better performance, which approaches the performance of the fourth case with a low-level loop gain of $\boldsymbol{K}_v = 32\frac{Nms}{rad}$ without reference adaptation. The second case with reference adaptation on only the x-direction has the same performance as the third case in this direction, while performing similar to the reference case in the y-direction. Following paragraphs will detail these effects by analysing the underlying control signals.

Figure 6.21 shows the velocity error in the task space $\boldsymbol{e}_{\dot{\boldsymbol{y}}} = \dot{\boldsymbol{y}}_{\boldsymbol{d}} - \boldsymbol{J}_{\boldsymbol{q}}\dot{\boldsymbol{q}}$.[§] The figure shows a fast decline in error in the directions with reference adaptation comparable in overshoot to a low-level loop gain of $\boldsymbol{K}_v = 32\frac{Nms}{rad}$ without reference adaptation. For the third case with reference adaptation only in the x-direction, the velocity error curve in this x-direction follows the corresponding velocity error curve of the second case with reference adaptation in all task space directions. The velocity error curve of the third case in the y-direction follows the corresponding velocity error curve of the reference case. *Hence the velocity error decreases in the directions with reference adaptation as if the robot had a larger proportional low-level velocity loop gain* $\boldsymbol{K}_v$*. However, for a similar*

[§]Remark that this error is not the same as the transformed joint velocity error $\boldsymbol{J}_{\boldsymbol{q}}\boldsymbol{e}_{\dot{\boldsymbol{q}}}$, which will be analysed in next paragraph. $\boldsymbol{e}_{\dot{\boldsymbol{y}}}$ indicates the real error in the task space, where $\boldsymbol{J}_{\boldsymbol{q}}\boldsymbol{e}_{\dot{\boldsymbol{q}}}$ contains the effect of the reference adaptation $\boldsymbol{B}''$, as can be seen in Figure 6.19.

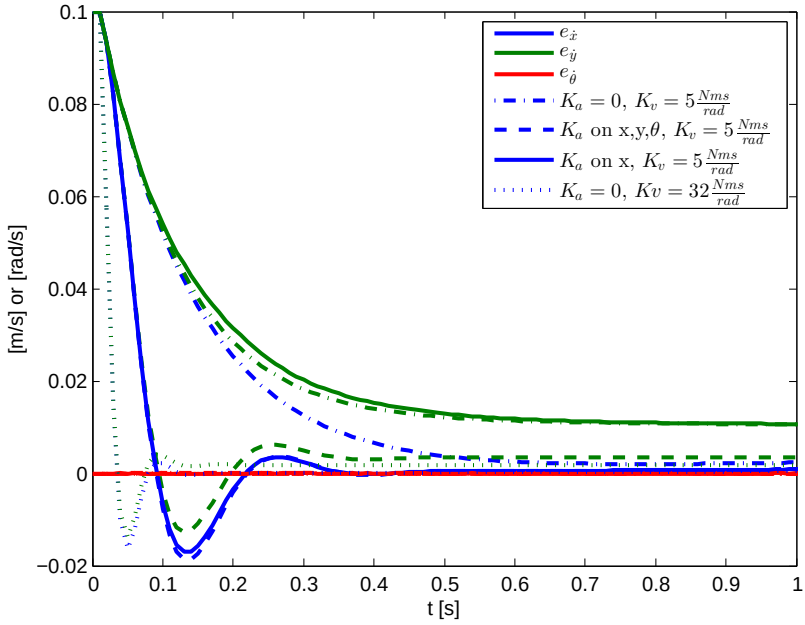


Figure 6.21: Velocity error in the task space $\boldsymbol{e_{\dot{y}}} = \dot{\boldsymbol{y}}_{ex} - \boldsymbol{J_q}\dot{\boldsymbol{q}}$. Different colors indicate the different task space directions, different line styles indicate different settings as indicated in the legend of the figure. $K_a = 0.7$ unless indicated as zero. The errors do not go to zero, but to a steady-state error (proportional first-order system).

error overshoot, the reaction is slower for the gain adapted directions than the larger proportional low-level velocity loop gain $\boldsymbol{K}_v$.

Figure 6.22 shows the joint velocity error transformed to the task space $\boldsymbol{J_q e_{\dot{q}}}$. This error forms an intermediate between the effect of the reference adaptation K_a and the low-level velocity loop gain $\boldsymbol{K}_v$. Equation 6.32 suggests that this error would be a *scaled version of* $\boldsymbol{e_{\dot{y}}}$ *for the directions with reference adaptation.* However, the equation is a result of an analysis in the continuous (Laplace) domain, while the implementation is in the discrete (Z-)domain. In this discrete domain, a loop without delay is impossible, hence a delay is introduced between $\boldsymbol{e_{\dot{q}}}$ and its transformation the task space. As a result, the effect of the reference adaptation loop has to build up at start-up, causing the *slower response* reported in Figure 6.21 and 6.22.

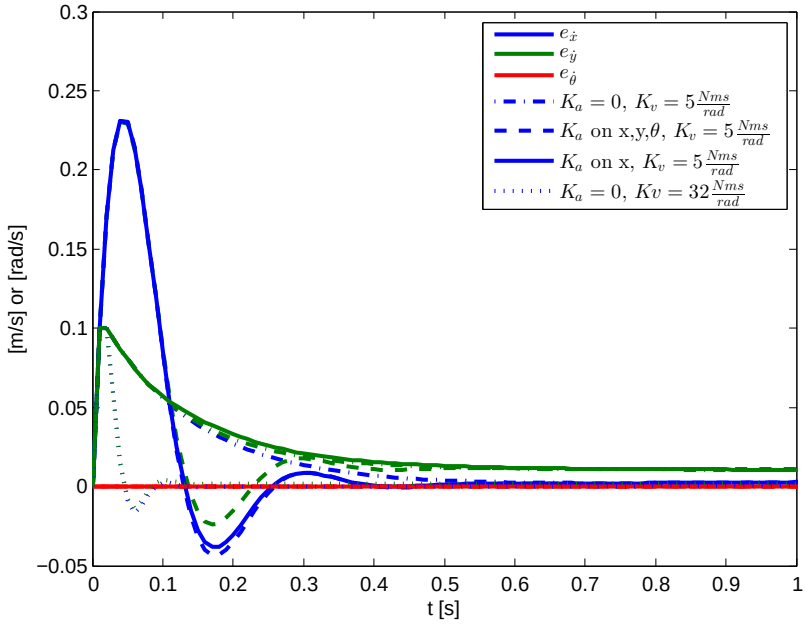


Figure 6.22: Joint velocity error transformed to the task space $\boldsymbol{J_q e_{\dot{q}}}$. Different colors indicate the different task space directions, different line styles indicate different settings as indicated in the legend of the figure. $K_a = 0.7$ unless indicated as zero. The errors do not go to zero, but to a steady-state error (proportional first-order system).

The desired joint velocity $\dot{\boldsymbol{q}}_{\boldsymbol{d}}$ has a transition phenomena in directions with reference adaptation as a result of the influence of the reference adaptation loop. Figure 6.23 depicts $\dot{\boldsymbol{q}}_{\boldsymbol{d}}$ over time, showing the transition phenomenon before these curves follow curves parallel to the directions without reference adaptation.

6.5.3 Discussion and conclusions

This section analysed and simulated a multi-dimensional first order velocity control scheme in joint space with task space reference adaptation. It considered only the free space motion, in order to study the effect of the task space reference adaptation loop.

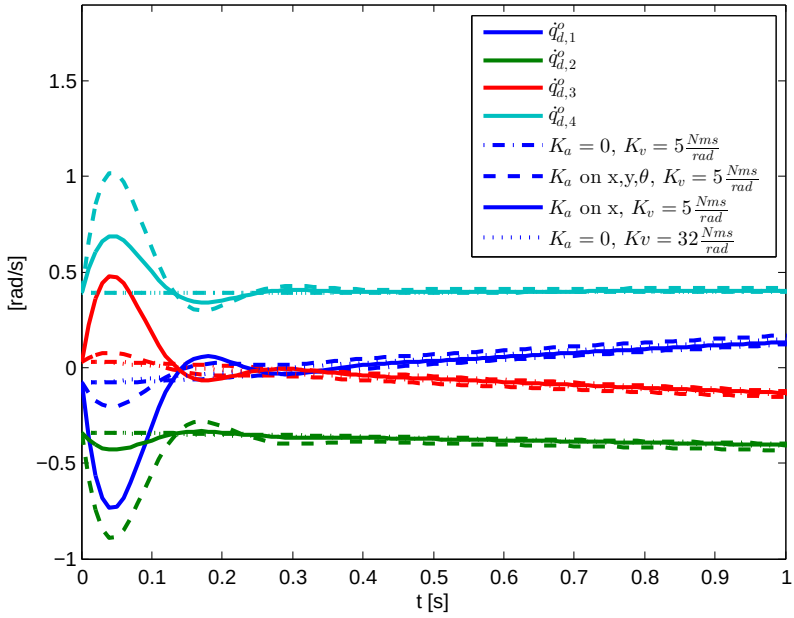


Figure 6.23: Desired joint velocity $\dot{\boldsymbol{q}}^{\circ}_{d}$. Different colors indicate the different robot joints, different line styles indicate different settings. $K_a = 0.7$ unless indicated as zero.

The analysis of the control scheme shows a similar, albeit multi-dimensional effect of the reference adaptation in the task space as in the one-DOF case analysed in previous section. **The simulation of an example case of a planar serial robot shows that the reference adaptation loop has the same effect as scaled low-level velocity loop gains, however it includes a delay due to the discrete nature of the implementation.** Applying the reference adaptation only in selected directions of the task space, results in a faster response in these directions as if the robot has higher velocity loop gains only in the selected directions. However, the reference adaptation in task space results in transition phenomenon in the joint space, which leads to higher motor torques. When designing the reference adaptation, it should be taken into account that these higher motor torques are within the limits of the motor capabilities.

6.6 Six-dimensional force-sensorless wrench control

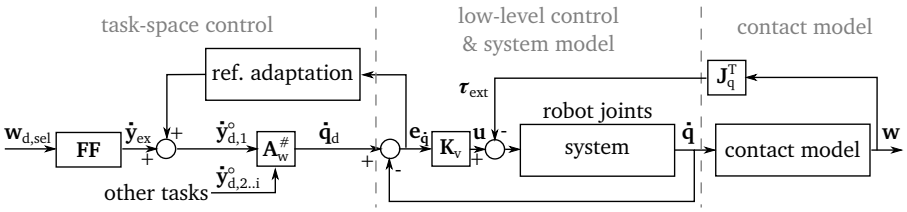


Figure 6.24: Part of the abstracted overview scheme for six-dimensional force-sensorless wrench control, discussed in Section 6.6.

This section analyses force-sensorless force control schemes for applications that require the robot to exert a certain (constant) wrench on the environment, without the need for high accuracy tracking or regulation performance. Figure 6.24 shows the abstracted scheme discussed in this section, it includes the feedforward factor ***FF*** and the reference adaptation loop in task space, other task space tasks, and a contact model.

The field of service robotics contains many examples of such applications including wiping a table, screwing a screw in a hole, or pushing a cart. All these examples have to simultaneously combine the ability to exert forces with other task constraints such as joint-limit and obstacle avoidance constraints. Therefore, the section analyses this combination of force-sensorless force control task constraints with other task constraints.

This section discusses table wiping as an example use case of the control schemes. In order to wipe a table (or a board), the robot has to first make contact with a plane, then excert a force on it, and last wipe the plane while maintaining the force on the plane. It is a classical use case of hybrid force-position control (discussed in Section 6.2), for which we present a resolved-resolved alternative within the iTaSC control scheme.

Table wiping is expressed the easiest as a set of two tasks in a single Cartesian task space between the table and the robot gripper: (i) The robot exerts a force on the table in a direction perpendicular to the table surface, defined as the z-direction of the task space; (ii) and the robot makes a movement parallel to the table surface while keeping the wiper aligned with the table. The latter defines constraints on the x, y-directions and the rotational DOF of the task space, respectively.

The control schemes presented in this section extend the one degree of freedom control schemes of Section 6.4 to the multi-dimensional case. To this end, the schemes integrate the multi-dimensional reference adaptation loop analysed in Section 6.5. As a result, this section combines and extends the assumptions made in previous sections, summarized here:

- The robot joints involved in the control scheme are **backdrivable**, and controlled by **velocity control loops with only proportional gains**. Therefore the joint velocity errors reflect the effects of possible joint-torque disturbances.
- These **low-level velocity controllers** run at a **sufficient high frequency** such that their interactions can be neglected. Therefore, the proportional gain matrix $\boldsymbol{K}_v$, the inertia matrix $\boldsymbol{I}_r$, and the damping matrix $\boldsymbol{C}_r$ are all diagonal matrices.
- The **gains of the low-level velocity controllers are known** but can or should not be altered.
- Unless specifically stated, the analysis considers only **under- but not over-constrained task spaces**. Therefore, the task space is of equal or lower dimensions than the joint space, and the augmented Jacobian $\boldsymbol{A}$ is of full row rank.
- The optimization problem is expressed and solved as a weighted, damped pseudo-inverse of the augmented Jacobian. Under abovementioned assumptions, the task weights will have no influence on the solution. Furthermore, the analysis assumes that the robot joints are far from singular configurations, hence the damping of the pseudo-inverse will not be activated. Therefore, the optimization problem reduces to the **pseudo-inverse of the augmented Jacobian**, with possibly weights $\boldsymbol{W}_q$ on the desired joint velocity.
- **The reference adaptation loop factor $\boldsymbol{K}_a$ is a diagonal matrix** with the diagonal elements not equal to 1.

The following subsections describe **two control schemes**. Both express the desired wrench $\boldsymbol{w}_d$ in a Cartesian space, however both transform this desired wrench to constraints in different spaces. The first control scheme transforms the desired wrench to **constraints in a Cartesian task space**, the second transforms the desired wrench to **constraints in joint (task) space**.[¶] The subsequent subsection discusses the differences between the two control schemes.

[¶]Joint space can also be a task space in this dissertation, meaning that the task is expressed in joint coordinates.

6.6.1 Control scheme

As mentioned above, the proposed control schemes express the wrench control constraints in a different task space, and hence result in a different optimization problem. The first control scheme, shown in Figure 6.25 and analysed in Section 6.6.2, expresses the constraints in a Cartesian task space; the second, shown in Figure 6.27 and analysed in Section 6.6.2, expresses the constraints in joint space. Both Figures 6.25 and 6.27 consist of three parts: (i) the left side models *task-space control*, (ii) the center models the *velocity-controlled robot system*, and (iii) the right side of the scheme models the *contact with the environment*.

The *task-space control* consists of the the set of tasks (constraints) imposed on the system and the resulting optimization problem. One of these tasks is the force-sensorless wrench control task. For the case of optimization in the Cartesian task space, the tasks other than the wrench control task are in the null space of this wrench control task. Since the case of optimization in the joint space constrains the whole robot joint space, any other task will conflict with the wrench control task. For both schemes of Figures 6.25 and 6.27 the force-sensorless control task consists of a feedforward term $\boldsymbol{FF}$ and the reference adaptation loop as explained in Section 6.5.

Both control schemes have as input to the control task a desired wrench $\boldsymbol{w}_d$ or part thereof $\boldsymbol{w}_{d,sel} = \boldsymbol{S}_c^{\#}\boldsymbol{w}_d$. The pseudo-inverse of the selection matrix $\boldsymbol{S}_c^{\#}$ selects the directions of the task space that will be constrained. For example a force is applied in the z-direction in the table wiping use case, hence $\boldsymbol{S}_c^{\#} = \boldsymbol{S}_c{}^T = [0\ 0\ 1\ 0\ 0\ 0]$. As will be detailed in the respective subsections, the feedforward term $\boldsymbol{FF}$ of both control schemes transform this input into different task spaces.

Each task results in a set of constraints that form an optimization problem that needs to be solved. The result of the optimization problem is a desired set of desired joint velocities.[‖] This chapter formulates the constraints as an augmented Jacobian $\boldsymbol{A}$ together with the desired output $\dot{\boldsymbol{y}}_d^{\circ}$. The (weighted) pseudo-inverse of the augmented Jacobian, $\boldsymbol{A}_W^{\#}$ solves the optimization problem.

Each row of the augmented Jacobian $\boldsymbol{A}$ relates a DOF of a task space with the joint space. Hence a task determines and constrains a set of rows of $\boldsymbol{A}$. We define the Jacobian of the wrench control task space as $\boldsymbol{J}_{\boldsymbol{q}1}$ of size $(n_{c,sel} \times n_q)$ and the other tasks as $\boldsymbol{J}_{\boldsymbol{q}2..i}$ of size $((n_c - n_{c,sel}) \times n_q)$, therefore $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{J}_{\boldsymbol{q}1} \\ \boldsymbol{J}_{\boldsymbol{q}2..i} \end{bmatrix}$. The selection matrix $\boldsymbol{S}$ of size $(n_c \times n_{c,sel})$, represents the selection of the

[‖] Section 2.2.3 gives a more detailed description on the optimization solving methods considered in this dissertation.

contribution of the wrench control task to the rows of $\boldsymbol{A}$, i.e. $\boldsymbol{J}_{\boldsymbol{q}1} = \boldsymbol{S}^{\#}\boldsymbol{A}$. In most cases, such as in the table wiping use case, $\boldsymbol{S}$ equals $\boldsymbol{S}_c$, however following sections discuss where these differ. For the wrench control task of the table wiping use case, the contribution of the force-sensorless wrench control task to the augmented Jacobian is the third row of the robot Jacobian $\boldsymbol{J}_{\boldsymbol{q}}$.

The model of the *velocity-controlled robot system*, in the center of Figures 6.25 and 6.27, regards only joint inertia $\boldsymbol{I}_r$ and damping $\boldsymbol{C}_r$. The gravity term $\boldsymbol{\tau}_g$ is left out of the equation, since the experiments use a gravity compensated robot. However, this term could be added in the model, and compensated for in the feedforward signal using a robot model and joint position measurements.

The *contact model*, on the right side of Figures 6.25 and 6.27, expresses the contact of the robot end effector with the environment in certain directions of the Cartesian output space indicated by the selection matrix $\boldsymbol{S}_c$. The contact is modelled as a set of springs with stiffness $\boldsymbol{K}_0$. The contact model and feedforward make use of the same selection matrix $\boldsymbol{S}_c$, in other words, the control schemes assume that the desired and actual contact directions are the same.

Following formulas describe the velocity-controlled robot system and contact model, which are common to both control schemes of Figures 6.25 and 6.27:

$$\dot{\boldsymbol{q}} = [\boldsymbol{I}_r s + \boldsymbol{C}_r]^{-1}[\boldsymbol{K}_v \boldsymbol{e}_{\dot{\boldsymbol{q}}} - \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{w}], \tag{6.33}$$

$$\boldsymbol{w} = \quad \boldsymbol{S}_c \boldsymbol{w}_c = \boldsymbol{S}_c \boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} \dot{\boldsymbol{q}}. \tag{6.34}$$

Following sections detail the control schemes of Figures 6.25 and 6.27.

6.6.2 Optimization in Cartesian task space

Figure 6.25 shows the control scheme with expression of the force-control task in a Cartesian task space.

Following equation defines the task space control part of Figure 6.25

$$\boldsymbol{e}_{\dot{\boldsymbol{q}}} = \dot{\boldsymbol{q}}_d - \dot{\boldsymbol{q}} = \boldsymbol{A}_W^{\#} \begin{bmatrix} \dot{\boldsymbol{y}}_{ex} + \boldsymbol{K}_a \boldsymbol{J}_{\boldsymbol{q}1} \boldsymbol{e}_{\dot{\boldsymbol{q}}} \\ \dot{\boldsymbol{y}}_{d,2..i}^{\circ} \end{bmatrix} - \dot{\boldsymbol{q}}. \tag{6.35}$$

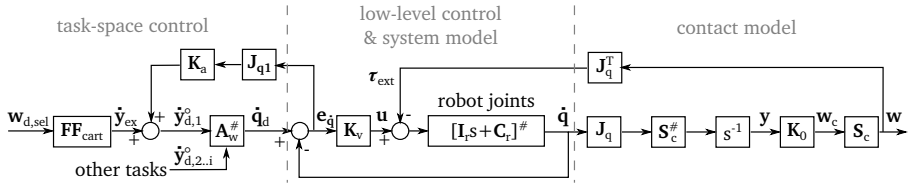


Figure 6.25: Multi-DOF control scheme expressing the force-control task constraints in Cartesian task space.

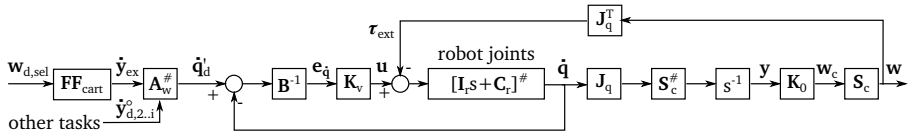


Figure 6.26: Equivalent multi-DOF control scheme expressing the force-control task constraints in Cartesian task space.

If the wrench control task constraints has priority over the other constraints, equation (6.35) can be written as

$$\begin{aligned} \boldsymbol{e}_{\dot{\boldsymbol{q}}} = \quad & \boldsymbol{J}_{\boldsymbol{q1}}^{\#}(\dot{\boldsymbol{y}}_{ex} + \boldsymbol{K}_a \boldsymbol{J}_{q1} \boldsymbol{e}_{\dot{\boldsymbol{q}}}) \\ + \quad & \tilde{\boldsymbol{J}}_{\boldsymbol{q2}}(\dot{\boldsymbol{y}}_{d,2..i}^{\circ} - \boldsymbol{J}_{\boldsymbol{q2}} \boldsymbol{J}_{\boldsymbol{q1}}^{\#}(\dot{\boldsymbol{y}}_{ex} + \boldsymbol{K}_a \boldsymbol{J}_{\boldsymbol{q1}} \boldsymbol{e}_{\dot{\boldsymbol{q}}})) - \dot{\boldsymbol{q}}. \end{aligned} \tag{6.36}$$

making use of equation (2.3) and

$$\tilde{\boldsymbol{J}}_{\boldsymbol{q2}} = (\boldsymbol{I} - \boldsymbol{J}_{\boldsymbol{q1}}^{\#} \boldsymbol{J}_{\boldsymbol{q1}})[\boldsymbol{J}_{\boldsymbol{q2}}(\boldsymbol{I} - \boldsymbol{J}_{\boldsymbol{q1}}^{\#} \boldsymbol{J}_{\boldsymbol{q1}})]^{\#}. \tag{6.37}$$

Under the assumptions listed in the introduction of this section, the constraints of lower priority do not conflict with the wrench control task constraints. Under these assumptions $\boldsymbol{J}_{\boldsymbol{q2}} \tilde{\boldsymbol{J}}_{\boldsymbol{q2}} = \boldsymbol{I}$ **. Therefore, they will result in the same solution as if they where all of the same priority.

Restructuring equation (6.36) results in

$$\boldsymbol{e}_{\dot{\boldsymbol{q}}} = \left[\boldsymbol{I} - (\boldsymbol{I} - \tilde{\boldsymbol{J}}_{\boldsymbol{q2}} \boldsymbol{J}_{\boldsymbol{q2}}) \boldsymbol{J}_{\boldsymbol{q1}}^{\#} \boldsymbol{K}_a \boldsymbol{J}_{\boldsymbol{q1}}\right]^{-1} \left[\boldsymbol{A}_W^{\#} \begin{bmatrix} \dot{\boldsymbol{y}}_{ex} \\ \dot{\boldsymbol{y}}_{d,2..i}^{\circ} \end{bmatrix} - \dot{\boldsymbol{q}}\right] = \boldsymbol{B}^{-1}[\boldsymbol{A}_W^{\#} \dot{\boldsymbol{y}}_d - \dot{\boldsymbol{q}}]. \tag{6.38}$$

The matrix $\boldsymbol{B}$, described by

$$\boldsymbol{B} = \boldsymbol{I} - (\boldsymbol{I} - \tilde{\boldsymbol{J}}_{\boldsymbol{q2}} \boldsymbol{J}_{\boldsymbol{q2}}) \boldsymbol{J}_{\boldsymbol{q1}}^{\#} \boldsymbol{K}_a \boldsymbol{J}_{\boldsymbol{q1}}, \tag{6.39}$$

** However, this does not necessarily imply that $\tilde{\boldsymbol{J}}_{\boldsymbol{q2}}$ equals $\boldsymbol{J}_{\boldsymbol{q2}}^{\#}$.

characterizes the effect of the reference adaptation loop on the controller. In effect, the gain $\boldsymbol{K}_v$ has been altered to the adapted gain $\boldsymbol{K}_b = \boldsymbol{K}_v \boldsymbol{B}^{-1}$, as represented in the equivalent scheme shown in Figure 6.26. Matrix $\boldsymbol{B}$ is of full rank and hence invertible if $\boldsymbol{K}_a \neq \boldsymbol{I}$.[††]

Feedforward

This section derives a $\boldsymbol{FF}_{cart}$ such that the wrench applied on the environment $\boldsymbol{w}_c$ equals the desired wrench $\boldsymbol{w}_{d,sel}$. The robot is in steady-state when the wrench applied by the robot is in equilibrium with the reaction wrench from the environment. In this case, the robot joint torques balance the reaction wrench, and the commanded robot torques equal the amplified force feedforward signal. In other words, the feedforward signal provides a system inversion feedforward signal which should cancel, in steady-state, the system transfer function. The derivation of the feedforward assumes the wrench control task is the only active task.

Solving equation (6.34) for $\dot{\boldsymbol{q}}$, given the desired wrench $\boldsymbol{w}_{d,sel}$ results in

$$\dot{\boldsymbol{q}} = (\boldsymbol{S}_c \boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}})^{\#} \boldsymbol{w}_d = (\boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}})^{\#} \boldsymbol{w}_{d,sel}. \tag{6.40}$$

Inserting this solution in equations (6.38) and (6.33) results in

$$\boldsymbol{e}_{\dot{\boldsymbol{q}}} = \boldsymbol{B}^{-1} \boldsymbol{J}_{\boldsymbol{q1}} \dot{\boldsymbol{y}}_{ex} - \boldsymbol{B}^{-1} (\boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}})^{\#} \boldsymbol{w}_{d,sel}, \tag{6.41}$$

$$(\boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}})^{\#} \boldsymbol{w}_{d,sel} = [\boldsymbol{I}_r s + \boldsymbol{C}_r]^{-1} [\boldsymbol{K}_v \boldsymbol{e}_{\dot{\boldsymbol{q}}} - \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{S}_c \boldsymbol{w}_{d,sel}]. \tag{6.42}$$

Combining equation (6.41) and (6.42) results in

$$\begin{aligned} &(\boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}})^{\#} \boldsymbol{w}_{d,sel} \\ &= \left[\boldsymbol{I}_r s + \boldsymbol{C}_r\right]^{-1} \left[\boldsymbol{K}_v \boldsymbol{B}^{-1} \boldsymbol{J}_{\boldsymbol{q1}} \dot{\boldsymbol{y}}_{ex} - \boldsymbol{K}_v \boldsymbol{B}^{-1} (\boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}})^{\#} \boldsymbol{w}_{d,sel} - \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{S}_c \boldsymbol{w}_{d,sel}\right]. \end{aligned} \tag{6.43}$$

Restructuring this result for $\boldsymbol{w}_{d,sel}$ returns

$$\left[(\boldsymbol{I}_r s + \boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}^{-1})(\boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}})^{\#} + \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{S}_c\right] \boldsymbol{w}_{d,sel} = \boldsymbol{K}_v \boldsymbol{B}^{-1} \boldsymbol{J}_{\boldsymbol{q1}} \dot{\boldsymbol{y}}_{ex}. \tag{6.44}$$

Hence the $\boldsymbol{FF}_{cart}$ that transforms a desired wrench $\boldsymbol{w}_{d,sel}$ to a task space velocity offset $\dot{\boldsymbol{y}}_{ex}$ becomes

$$\boldsymbol{FF}_{cart} = \boldsymbol{J}_{\boldsymbol{q1}}^{\#} \boldsymbol{B} \boldsymbol{K}_v^{-1} \left[(\boldsymbol{I}_r s + \boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}^{-1})(\boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}})^{\#} + \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{S}_c\right] \tag{6.45}$$

[††] Appendix A.1 gives a proof.

When regarding only steady-state, equation (6.45) becomes

$$\begin{aligned} \boldsymbol{FF}_{cart}|_{s=0} &= \boldsymbol{J_{q1}} \boldsymbol{B} {\boldsymbol{K_v}}^{-1} \boldsymbol{J_q}^T \boldsymbol{S}_c \\ &= \boldsymbol{J_{q1}} (\boldsymbol{I} - \boldsymbol{J_{q1}}^{\#} \boldsymbol{K}_a \boldsymbol{J_{q1}}) {\boldsymbol{K_v}}^{-1} \boldsymbol{J_q}^T \boldsymbol{S}_c \\ &= (\boldsymbol{I} - \boldsymbol{K}_a) \boldsymbol{J_{q1}} {\boldsymbol{K_v}}^{-1} \boldsymbol{J_q}^T \boldsymbol{S}_c \end{aligned} \tag{6.46}$$

This expression can be interpreted as a Cartesian task space (or operational space) compliance. Remark that the feedforward matrix $\boldsymbol{FF}_{cart}|_{s=0}$ is dependent on the known low-level control gains, but independent of the motor and contact model. The latter disappear due to the integration in the contact model (from velocity to position). The equation first transforms the desired wrench to desired joint torques using the Jacobian transpose $\boldsymbol{J_q}^T$. Then it uses the adapted gain of the low-level controller, $\boldsymbol{B}^{-1}\boldsymbol{K}_v$ and ${\boldsymbol{B}'_I}^{-1}\boldsymbol{K}_v$ respectively, which represents the damping with respect to the desired force, resulting in the desired joint velocities. Finally, in case of $\boldsymbol{FF}_{cart}$, these desired joint velocities are projected to a selected part of a Cartesian task space.

For the use case of table wiping, the feedforward matrix transforms the desired force in the z-direction to a desired velocity in the same direction.

Transfer function

This section deduces the transfer function $\boldsymbol{G}_{cart}$ from the task space velocity offset $\dot{\boldsymbol{y}}_{ex}$ to the resulting wrench applied on the environment $\boldsymbol{w}_c$. Remark that the derivation of the transfer function is not the pseudo-inverse of the feedforward term: the derivation of this feedforward searches for *one of the possible* $\dot{\boldsymbol{q}}$ that will result in the force/torque constriants, as expressed in equation (6.40); the derivation of the transfer function is a forward analysis, expressing the $\boldsymbol{w}_c$ that results from a specific $\dot{\boldsymbol{q}}$.

Inserting equation (6.38) in equation (6.33) and solving for $\dot{\boldsymbol{q}}$ results in

$$\dot{\boldsymbol{q}} = [\boldsymbol{I}_r s + \boldsymbol{C}_r]^{-1} [\boldsymbol{K}_v \boldsymbol{B}^{-1} (\boldsymbol{A}_W^{\#} \dot{\boldsymbol{y}}_d - \dot{\boldsymbol{q}}) - \boldsymbol{J_q}^T \boldsymbol{w}], \tag{6.47}$$

$$\dot{\boldsymbol{q}} = [\boldsymbol{I}_r s + \boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}^{-1}]^{-1} [\boldsymbol{K}_v \boldsymbol{B}^{-1} \boldsymbol{A}_W^{\#} \dot{\boldsymbol{y}}_d - \boldsymbol{J_q}^T \boldsymbol{w}]. \tag{6.48}$$

Further inserting the result of equation (6.48) in equation (6.34), and grouping $\boldsymbol{w}_c$ dependent terms, results in

$$\left[\boldsymbol{I} + \boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} (\boldsymbol{I}_r s + \boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}^{-1})^{-1} \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{S}_c\right] \boldsymbol{w}_c$$

$$= \boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} [\boldsymbol{I}_r s + \boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}^{-1}]^{-1} \boldsymbol{K}_v \boldsymbol{B}^{-1} \boldsymbol{A}_W^{\#} \dot{\boldsymbol{y}}_d. \quad (6.49)$$

Assuming the multiplication factor of $\boldsymbol{w}_c$ is of full rank, this results in the transfer function

$$\boldsymbol{G}_{cart} = \left[\boldsymbol{I} + \boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} (\boldsymbol{I}_r s + \boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}^{-1})^{-1} \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{S}_c\right]^{-1}$$

$$\boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} [\boldsymbol{I}_r s + \boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}^{-1}]^{-1} \boldsymbol{K}_v \boldsymbol{B}^{-1} \boldsymbol{A}_W^{\#}, \quad (6.50)$$

such that $\boldsymbol{w}_c = \boldsymbol{G}_{cart} \dot{\boldsymbol{y}}_d$. When only regarding steady-state and assuming a full rank environment stiffness matrix $\boldsymbol{K}_0$, equation (6.50) reduces to

$$\boldsymbol{G}_{cart}|_{s=0} = \left[\boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} (\boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}^{-1})^{-1} \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{S}_c\right]^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} [\boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}^{-1}]^{-1} \boldsymbol{K}_v \boldsymbol{B}^{-1} \boldsymbol{A}_W^{\#}. \quad (6.51)$$

This equation reflects the effect of the robot joint dynamics and the adapted velocity controller. The effect of the total damping $\boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}^{-1}$ will only be eliminated in case of a fully force constrained ($\boldsymbol{S}_c = \boldsymbol{I}$), non-redundant robot in a non-singular configuration. Remark that both primary and secondary task constraints influence the wrench applied on the environement $\boldsymbol{w}_c$.

Applying the feedforward to the system

Applying the feedforward $\boldsymbol{FF}_{cart}$ to the velocity-controlled robot system and assuming steady-state, results in

$$\begin{aligned} \boldsymbol{w}_c &= \boldsymbol{G}_{cart}|_{s=0} \begin{bmatrix} \boldsymbol{FF}_{cart} & \boldsymbol{w}_{d,sel} \\ & \dot{\boldsymbol{y}}_{d,2..i}^{\circ} \end{bmatrix} \\ &= \left[\boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} (\boldsymbol{C}_r + \boldsymbol{K}_b)^{-1} \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{S}_c\right]^{-1} \\ &\quad \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} (\boldsymbol{C}_r + \boldsymbol{K}_b)^{-1} \boldsymbol{K}_b \boldsymbol{A}_W^{\#} \begin{bmatrix} \boldsymbol{J}_{\boldsymbol{q}1} \boldsymbol{K}_b{}^{-1} \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{S}_c \boldsymbol{w}_{d,sel} \\ \dot{\boldsymbol{y}}_{d,2..i}^{\circ} \end{bmatrix}. \end{aligned} \quad (6.52)$$

Assuming that the gains of the joint velocity controller are significantly bigger than the robot joint damping, i.e. $\boldsymbol{K}_b >> \boldsymbol{C}_r$, $\boldsymbol{C}_r$ can be neglected. Moreover,

assuming that contact is made in the foreseen directions $\boldsymbol{S} = \boldsymbol{S}_c$, and since the constraints of the other tasks do not conflict with the wrench control constraints, $\boldsymbol{S}_c^{\#}\boldsymbol{J}_{\boldsymbol{q}}\boldsymbol{A}_W^{\#} = \boldsymbol{I}$ and $\boldsymbol{S}_c^{\#}\boldsymbol{J}_{\boldsymbol{q}} = \boldsymbol{S}^{\#}\boldsymbol{J}_{\boldsymbol{q}} = \boldsymbol{J}_{\boldsymbol{q1}}$. Integrating this assumptions in equation 6.52 results in

$$\boldsymbol{w}_c = \left[\boldsymbol{J}_{\boldsymbol{q1}}\boldsymbol{K}_b{}^{-1}\boldsymbol{J}_{\boldsymbol{q}}^T\boldsymbol{S}_c\right]^{-1}\boldsymbol{J}_{\boldsymbol{q1}}\boldsymbol{K}_b{}^{-1}\boldsymbol{J}_{\boldsymbol{q}}^T\boldsymbol{S}_c\boldsymbol{w}_{d,sel} = \boldsymbol{w}_{d,sel}. \tag{6.53}$$

However, in case the robot joint damping $\boldsymbol{C}_r$ is not negligeable with respect to the gain K_b, and in case the wrench control task constraints are the only active constraints, equation 6.52 reduces to

$$\begin{aligned}\boldsymbol{w}_c = &\left[\boldsymbol{S}_c^{\#}\boldsymbol{J}_{\boldsymbol{q}}(\boldsymbol{C}_r + \boldsymbol{K}_b)^{-1}\boldsymbol{J}_{\boldsymbol{q}}^T\boldsymbol{S}_c\right]^{-1} \\ &\boldsymbol{S}_c^{\#}\boldsymbol{J}_{\boldsymbol{q}}(\boldsymbol{C}_r + \boldsymbol{K}_b)^{-1}\boldsymbol{K}_b\boldsymbol{J}_{\boldsymbol{q1}}^{\#}\boldsymbol{J}_{\boldsymbol{q1}}\boldsymbol{K}_b{}^{-1}\boldsymbol{J}_{\boldsymbol{q}}^T\boldsymbol{S}_c\boldsymbol{w}_{d,sel}.\end{aligned} \tag{6.54}$$

In this case, $\boldsymbol{w}_c$ equals $\boldsymbol{w}_{d,sel}$ when $\boldsymbol{J}_{\boldsymbol{q1}}^{\#}\boldsymbol{J}_{\boldsymbol{q1}} = \boldsymbol{I}$. However, $\boldsymbol{J}_{\boldsymbol{q1}}^{\#}\boldsymbol{J}_{\boldsymbol{q1}}$ is an orthogonal projection matrix in the column space of $\boldsymbol{J}_{\boldsymbol{q1}}$. It equals the unity matrix only in case of a fully force constraint, non-redundant robot.

This projection error cannot be compensated in the Cartesian task space, seen its lower dimensionality than the robot joint space. Next Section 6.6.3 introduces a control scheme without this projection error, by avoiding the transformation to a lower dimensional task space.

This analysis adds a new requirement to the list of requirements stated in Section 6.4.2. The choice of K_a should result in a value of $\boldsymbol{K}_b$ such that $\boldsymbol{K}_b >> \boldsymbol{C}_r$.

6.6.3 Optimization in joint space

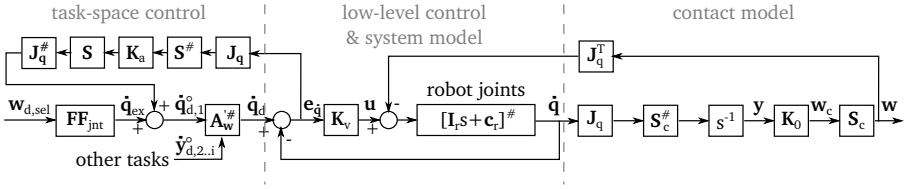


Figure 6.27: Multi DOF control scheme expressing the force-control task constraints in the joint (configuration) space

The analysis in Section 6.6.2 derives a feedforward term expressed by equation (6.46). This equation transforms a desired wrench $\boldsymbol{w}_d$ to desired

robot joint velocities, and transforms these robot joint velocities on their turn to Cartesian task space velocity constraints. The optimization problem transforms these (and other) constraints back to the joint space. As the analysis of Section 6.6.2 shows, this projection in the columnspace of the robot jacobian $\boldsymbol{J_q}$ results in errors in the resulting wrench applied on the environment $\boldsymbol{w}_c$.

In contrast, **this section analyses a similar control scheme that avoids this projection. The desired wrench $\boldsymbol{w}_d$ is still expressed in a Cartesian task space, as is the reference adaptation. However, both are transformed to constraints in the joint space.**

Figure 6.27 shows the control scheme expressing this force-control with constraints in the joint space. The figure is similar to Figure 6.25, but the force-sensorless wrench control task consists here of a feedforward term and a reference adaptation loop resulting in a set of desired joint velocities $\dot{\boldsymbol{q}}^o_{d,1}$. As a consequence the part of the $\boldsymbol{A}$ matrix related to the wrench control task is the unity matrix, its weighted, pseudo-inverse is denoted $\boldsymbol{A}^{'\#}_{\boldsymbol{W}}$, the apostrophe marks the difference with the scheme of Section 6.6.2. Since the wrench control task applies constrains the whole robot joint space, the null space of its Jacobian is empty. In other words, any other task will conflict with the wrench control task. This conflict can be influenced using task weights. *This section analyses the case where the wrench control task is the only task*, in this case $\boldsymbol{A}^{'\#}_{\boldsymbol{W}}$ equals the identity matrix.

The controlled system remains the same as in previous section, hence the equation describing the output of the low-level control loop and system model, equation 6.33, as well as the equation describing the output of the contact model, equation 6.34, still hold. Equation 6.35 changes to

$$\boldsymbol{e}_{\dot{\boldsymbol{q}}} = \dot{\boldsymbol{q}}_{ex} + \boldsymbol{J}^{\#}_{\boldsymbol{q}} \boldsymbol{S} \boldsymbol{K}_a \boldsymbol{S}^{\#} \boldsymbol{J}_{\boldsymbol{q}} \boldsymbol{e}_{\dot{\boldsymbol{q}}} - \dot{\boldsymbol{q}}. \tag{6.55}$$

Remark the use of $\boldsymbol{J}^{\#}_{\boldsymbol{q}} \boldsymbol{S}$ and not $(\boldsymbol{S}^{\#} \boldsymbol{J}_{\boldsymbol{q}})^{\#}$. Both will result in the same reference adaptation in the selected directions, however, the chosen option will leave the gain unadapted in the other Cartesian directions. This can also be easily seen by regarding $\boldsymbol{S} \boldsymbol{K}_a \boldsymbol{S}^{\#}$, which explicitly puts the reference adaptation of non-selected direction to zero.

Restructuring equation 6.55 and introducing $\boldsymbol{B}' = \boldsymbol{I} - \boldsymbol{J}^{\#}_{\boldsymbol{q}} \boldsymbol{S} \boldsymbol{K}_a \boldsymbol{S}^{\#} \boldsymbol{J}_{\boldsymbol{q}}$ (full rank if $\boldsymbol{K}_a \neq 1$) results in

$$\boldsymbol{e}_{\dot{\boldsymbol{q}}} = [\boldsymbol{I} - \boldsymbol{J}^{\#}_{\boldsymbol{q}} \boldsymbol{S} \boldsymbol{K}_a \boldsymbol{S}^{\#} \boldsymbol{J}_{\boldsymbol{q}}]^{\#} [\dot{\boldsymbol{q}}_{ex} - \dot{\boldsymbol{q}}] = \boldsymbol{B}'^{-1} [\dot{\boldsymbol{q}}_{ex} - \dot{\boldsymbol{q}}]. \tag{6.56}$$

$\boldsymbol{B}'$ represents the effect of the feedback loop with the $\boldsymbol{K}_a$ factor on the gain $\boldsymbol{K}_v$, resulting in the adapted gain $\boldsymbol{K}_v{}' = \boldsymbol{K}_v \boldsymbol{B}'^{-1}$. Remark the similarity between $\boldsymbol{B}'$ and $\boldsymbol{B}$ of previous section. Both will have the same reference adaptation effect in the selected task directions.

Feedforward

This section derives a $\boldsymbol{FF}_{jnt}$ such that the wrench applied on the environment $\boldsymbol{w}_c$ equals the desired wrench $\boldsymbol{w}_{d,sel}$. The analysis starts from equation (6.40), which relates the joint velocities $\dot{\boldsymbol{q}}$ with the desired contact wrench $\boldsymbol{w}_c = \boldsymbol{w}_{d,sel}$ using the contact model. Inserting this equation in equation 6.56 results in

$$\boldsymbol{e}_{\dot{\boldsymbol{q}}} = \boldsymbol{B}'^{-1}\dot{\boldsymbol{q}}_{ex} - \boldsymbol{B}'^{-1}(\boldsymbol{K}_0 s^{-1}\boldsymbol{S}_c^{\#}\boldsymbol{J}_{\boldsymbol{q}})^{\#}\boldsymbol{w}_{d,sel}. \tag{6.57}$$

Combining equation (6.57) and (6.42) results in

$$(\boldsymbol{K}_0 s^{-1}\boldsymbol{S}_c^{\#}\boldsymbol{J}_{\boldsymbol{q}})^{\#}\boldsymbol{w}_{d,sel} = \left[\boldsymbol{I}_r s + \boldsymbol{C}_r\right]^{-1}\left[\boldsymbol{K}_v\boldsymbol{B}'^{-1}\dot{\boldsymbol{q}}_{ex} - \boldsymbol{K}_v\boldsymbol{B}'^{-1}(\boldsymbol{K}_0 s^{-1}\boldsymbol{S}_c^{\#}\boldsymbol{J}_{\boldsymbol{q}})^{\#}\boldsymbol{w}_{d,sel} - \boldsymbol{J}_{\boldsymbol{q}}^T\boldsymbol{S}_c\boldsymbol{w}_{d,sel}\right]. \tag{6.58}$$

Restructuring this result for $\boldsymbol{w}_{d,sel}$ returns

$$\left[(\boldsymbol{I}_r s + \boldsymbol{C}_r + \boldsymbol{K}_v\boldsymbol{B}'^{-1})(\boldsymbol{K}_0 s^{-1}\boldsymbol{S}_c^{\#}\boldsymbol{J}_{\boldsymbol{q}})^{\#} + \boldsymbol{J}_{\boldsymbol{q}}^T\boldsymbol{S}_c\right]\boldsymbol{w}_{d,sel} = \boldsymbol{K}_v\boldsymbol{B}'^{-1}\dot{\boldsymbol{q}}_{ex}. \tag{6.59}$$

Hence the $\boldsymbol{FF}_{joint}$ which transforms a desired wrench $\boldsymbol{w}_{d,sel}$ to a task space velocity offset $\dot{\boldsymbol{q}}_{ex}$ becomes

$$\boldsymbol{FF}_{joint} = \boldsymbol{B}'\boldsymbol{K}_v^{-1}\left[(\boldsymbol{I}_r s + \boldsymbol{C}_r + \boldsymbol{K}_v\boldsymbol{B}'^{-1})(\boldsymbol{K}_0 s^{-1}\boldsymbol{S}_c^{\#}\boldsymbol{J}_{\boldsymbol{q}})^{\#} + \boldsymbol{J}_{\boldsymbol{q}}^T\boldsymbol{S}_c\right] \tag{6.60}$$

When regarding only steady-state, equation (6.60) becomes

$$\boldsymbol{FF}_{joint}|_{s=0} = \boldsymbol{B}'\boldsymbol{K}_v^{-1}\boldsymbol{J}_{\boldsymbol{q}}^T\boldsymbol{S}_c, \tag{6.61}$$

which equals the expression of $\boldsymbol{FF}_{cart}|_{s=0}$, but without the Cartesian task space transformation.

Transfer function

This section deduces the transfer function $\boldsymbol{G}_{joint}$ from the task space velocity offset $\dot{\boldsymbol{q}}_{ex}$ to the resulting wrench applied on the environment $\boldsymbol{w}_c$.

Inserting equation (6.56) in equation (6.33) and solving for $\dot{\boldsymbol{q}}$ results in

$$\dot{\boldsymbol{q}} = \left[\boldsymbol{I}_r s + \boldsymbol{C}_r\right]^{-1}\left[\boldsymbol{K}_v\boldsymbol{B}'^{-1}(\dot{\boldsymbol{q}}_{ex} - \dot{\boldsymbol{q}}) - \boldsymbol{J}_{\boldsymbol{q}}^T\boldsymbol{w}\right], \tag{6.62}$$

$$\dot{\boldsymbol{q}} = [\boldsymbol{I}_r s + \boldsymbol{C}_r + \boldsymbol{K}_v\boldsymbol{B}'^{-1}]^{-1}[\boldsymbol{K}_v\boldsymbol{B}'^{-1}\dot{\boldsymbol{q}}_{ex} - \boldsymbol{J}_{\boldsymbol{q}}^T\boldsymbol{w}]. \tag{6.63}$$

Further inserting the result of equation (6.63) in equation (6.34), and grouping $\boldsymbol{w}_c$ dependent terms, results in

$$\left[\boldsymbol{I} + \boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} (\boldsymbol{I}_r s + \boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}'^{-1})^{-1} \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{S}_c\right] \boldsymbol{w}_c$$
$$= \boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} [\boldsymbol{I}_r s + \boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}'^{-1}]^{-1} \boldsymbol{K}_v \boldsymbol{B}'^{-1} \dot{\boldsymbol{q}}_{ex}. \quad (6.64)$$

Assuming the multiplication factor of $\boldsymbol{w}_c$ is full rank, this results in the transfer function

$$\boldsymbol{G}_{joint} = \left[\boldsymbol{I} + \boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} (\boldsymbol{I}_r s + \boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}'^{-1})^{-1} \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{S}_c\right]^{-1}$$
$$\boldsymbol{K}_0 s^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} [\boldsymbol{I}_r s + \boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}'^{-1}]^{-1} \boldsymbol{K}_v \boldsymbol{B}'^{-1}, \quad (6.65)$$

such that $\boldsymbol{w}_c = \boldsymbol{G}_{joint} \dot{\boldsymbol{q}}_{ex}$. When only regarding steady-state and assuming a full rank environment stiffness matrix $\boldsymbol{K}_0$, equation (6.50) reduces to

$$\boldsymbol{G}_{joint}|_{s=0} = \left[\boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} (\boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}'^{-1})^{-1} \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{S}_c\right]^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} [\boldsymbol{C}_r + \boldsymbol{K}_v \boldsymbol{B}'^{-1}]^{-1} \boldsymbol{K}_v \boldsymbol{B}'^{-1}. \quad (6.66)$$

This equation (6.66) is exactly the same equation as equation (6.51), but with the replacement of $\boldsymbol{B}$ by $\boldsymbol{B}'$, and without the Cartesian to joint space transformation $\boldsymbol{A}_W^{\#}$.

Applying the feedforward to the system

Applying the feedforward $\boldsymbol{FF}_{jnt}$ to the velocity-controlled robot system and assuming steady-state, results in

$$\begin{aligned} \boldsymbol{w}_c &= \boldsymbol{G}_{jnt}|_{s=0} \quad \boldsymbol{FF}_{joint} \quad \boldsymbol{w}_{d,sel} \\ &= \left[\boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} (\boldsymbol{C}_r + \boldsymbol{K}_b')^{-1} \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{S}_c\right]^{-1} \boldsymbol{S}_c^{\#} \boldsymbol{J}_{\boldsymbol{q}} [\boldsymbol{C}_r + \boldsymbol{K}_b']^{-1} \boldsymbol{K}_b' \boldsymbol{K}_b'^{-1} \boldsymbol{J}_{\boldsymbol{q}}^T \boldsymbol{S}_c \boldsymbol{w}_{d,sel} \\ &= \boldsymbol{w}_{d,sel}. \end{aligned} \quad (6.67)$$

Since this subsection assumes that only the wrench control task is active, there is no effect of other constraints. Comparing the combined transfer function of equation (6.67) with the similar Cartesian resolved equation (6.54), shows two differences. First, the reference adaptation loop, and hence the adapted gain $\boldsymbol{K}_b$ are different. The effect in the wrench controlled task directions will

however be same. Second, the orthogonal projection matrix $\boldsymbol{J}_{\boldsymbol{q1}}^{\#}\boldsymbol{J}_{\boldsymbol{q1}}$ does not appear, as intended in the design of the control scheme.

The resulting wrench is independent from the gain $\boldsymbol{K}_b{}'$. Therefore, and in contrast to the Cartesian resolved case, the reference adaptation loop will have no influence on the steady-state result. In practice, it is still desirable to have a sufficient large gain $\boldsymbol{K}_b{}'$ to be more robust against the here omitted disturbances.

The theoretical analysis shows that the applied wrench equals the desired wrench in steady-state, in case the wrench control task is the only active task. Adding other constraints of the same or higher priority, will conflict with the wrench control task, which spans the whole robot joint space. This conflict will cause a deviation from the desired $\boldsymbol{w}_{d,sel}$.

For example the position control task of the presented use case of robotic table wiping will not be feasible when of lower priority, or will be weighted with the wrench control task. This weighting on velocity level results in an output without useful meaning in the context of the position or wrench control task.

6.6.4 Conclusions

This section describes two variations of a force-sensorless wrench control scheme. Both express (part of) a desired wrench in a Cartesian space, however both transform this desired wrench to constraints in different spaces.

The first control scheme, shown in Figure 6.25, transforms the desired wrench to **constraints in a Cartesian task space**. This control scheme allows the combination of the wrench control task with other constraints in its null space. The applied wrench will be equal to the desired wrench if the robot joint damping $\boldsymbol{C}_r$ can be neglected with respect to the adapted low-level velocity gain $\boldsymbol{K}_b$, i.e. $\boldsymbol{K}_b >> \boldsymbol{C}_r$. As a result, the choice of the reference adaptation factor $\boldsymbol{K}_a$ is a trade-off between a sufficiently high $\boldsymbol{K}_b$, and a non-overdamped system, as stated in Section 6.4.2. In case (i) the robot joint damping is non-negligible, (ii) the robot has redundant degrees-of-freedom, and (iii) the wrench control task constraints are the only active constraints, the applied wrench is distorted, characterized by the orthogonal projection matrix in the column space of $\boldsymbol{J}_{\boldsymbol{q1}}$. This error cannot be compensated in the Cartesian task space, because of its lower dimensionality than the robot joint space.

The second control scheme, shown in Figure 6.27, transforms the desired wrench to **constraints in joint (task) space**. This control scheme avoids the error caused by the orthogonal projection matrix in the first control scheme, by avoiding the transformation to a lower dimensional task space. In contrast

to the Cartesian resolved case, the resulting wrench is independent from the gain $\boldsymbol{K}_b{}'$. As a result, the reference adaptation loop will have no influence on the steady-state result. Nevertheless, it is still necessary to have a sufficient large gain $\boldsymbol{K}_b{}'$ to be more robust against disturbances. However, adding other constraints of the same or higher priority to the wrench control task, will conflict with the wrench control task since the latter constrains the whole robot joint space. This conflict will cause a deviation from the desired $\boldsymbol{w}_{d,sel}$.

The analysis above hints following **usage pattern**:

- In case the force-sensorless wrench control task is the only active task use the scheme of Figure 6.27 that expresses the force constraints in joint space, especially if, in addition, the robot joint damping cannot be neglected.
- In case the force-sensorless wrench control task is not the only active task, use the scheme of Figure 6.25 which expresses the force constraints in Cartesian task space.

Note that next to the mentioned sources of deviation, also unmodeled effects or model errors will add to the deviation of the applied wrench $\boldsymbol{w}_c$ with respect to the desired wrench $\boldsymbol{w}_{d,sel}$.

6.7 Experimental validation and results

This section validates the control schemes introduced in Section 6.6 using a PR2 robot. The same value of K_a is chosen for all task directions, hence $\boldsymbol{K}_a$ reduces to K_a. A **first set of experiments** (Section 6.7.2) compares the performance of the robot applying a force on a table for different values of K_a. From these experiments an appropriate value of K_a is chosen, which is used throughout the section.

A **second set of experiments** compares the performance of the control scheme that expresses the force-control task constraints in Cartesian task space, versus the control schemes that expresses these constriants in joint space. Therefore, the experiments compare the performance of the robot applying a force in different directions: first in a single direction (Section 6.7.3), then in multiple directions at the same time (Section 6.7.4).

A **third set of experiments** analyses the use case of table wiping (Section 6.7.5).

Section 6.7.6 summarizes and compares the results of the different experiments.

Appendix C.3 lists and discusses videos of the experiments, which are made available online [164].

6.7.1 Experimental setup

The experiments feature a PR2 robot that applies a force in different directions of a Cartesian task space.

The robot uses the seven joints of its left arm to apply a force with the tip of its gripper on a force sensor that is attached to the environment. The experiments use the force sensor only for external verification, its data is not used in the control loops. Figure 6.28 shows the setup in case of applying a force on a table, with indication of the Cartesian task space within the iTaSC framework. This feature space spans the space between the object frame defined on the contact point on the force sensor $\{o1\}$, and the object frame on the left gripper $\{o2\}$.

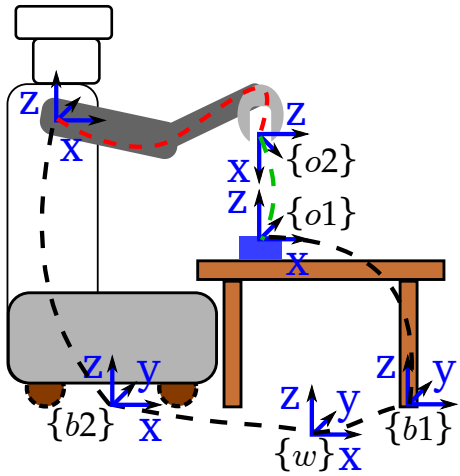


Figure 6.28: Setup of the experiment where the PR2 robot has to apply a force on the force sensor (blue) attached on the table. The dashed lines indicate the kinematic loop used in the force control task: black lines indicate fixed kinematic relations, red lines indicate the robot joints controlled by the low-level velocity controller, and green lines indicate the force control task space on which constraints are imposed.

Two main directions are considered, (i) the vertical z-direction, for example when the robot applies a force on a table, and (ii) the horizontal y-direction, for example when the robot applies a force on a wall.

The different experiments will consider the combination of one or more of following tasks:

- *A wrench control task* applies a (part of) a desired wrench by expressing the task constraints in Cartesian or joint task space.
- *A pose control task* regulates the non-wrench controlled directions of the Cartesian task space to a desired setpoint using a PI-controller. Rotations are grouped in a rotation matrix expression in order to avoid representation singularities.
- *A joint configuration task* regulates the arm to a desired robot joint configuration with a proportional controller. This task is a secondary constraint, i.e. in the null space of the other active tasks. The desired configuration is pre-defined for each setup, it consists of joint positions far from joint limits or singular arm configurations.

Since there are no conflicting constraints within one task priority, no task weighting is applied. The robot joints are weighted, penalizing the use of the heavy shoulder joint and the wrist joints, which have limited backdrivability.

The experiment design assures that the commanded motor torques are within maximal bounds, and that the robot joints remain far from singularities and joint limits. The smallest singular value of the calculation of the pseudo-inverse of the augmented Jacobian has a numerical value of 0.9. The pseudo-inverse needs no damping throughout the experiments.

Each of the experiments is repeated using the same setup, and is retaken for different contact points in the robot workspace. Figures 6.29 and 6.30 show the location of these points for the experiments in the z- and y-direction, respectively.

Figure 6.31 shows the experimental setup for the experiments in the y-direction, and combined y- and z-direction.

The section will show boxplots of the measurement data. A box of a boxplot represents information on a set of ten repeated experiments in a certain contact point. There are three different contact points, each represented by a different box color. Each box consists of (i) a central mark, indicating the median; (ii) box edges, indicating the 25th and 75th percentiles; (iii) the wiskers, indicating the most extreme data points not considered outliers; and (iv) red crosses, indicating eventual outliers. The interquartile range is defined as the difference between the 25th and 75th percentiles. Measurement more than one and a half times the interquartile range removed from the 25th or 75th percentile are considered outliers.

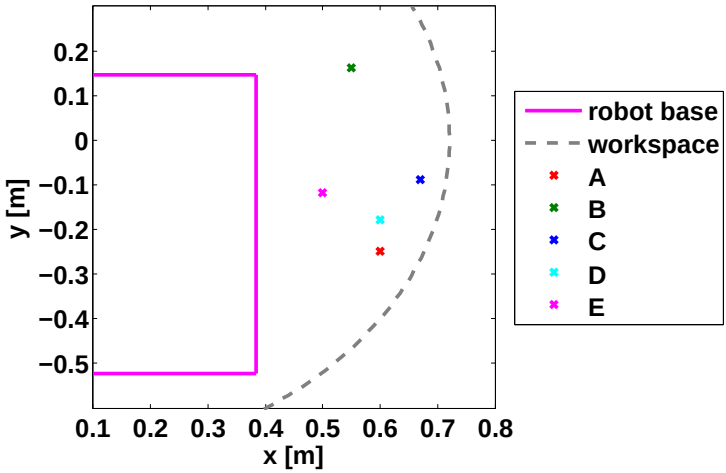


Figure 6.29: Location of the contact points in the robot workspace for experiments in the z-direction. The grey dashed line indicates the approximate boundary of the robot workspace.

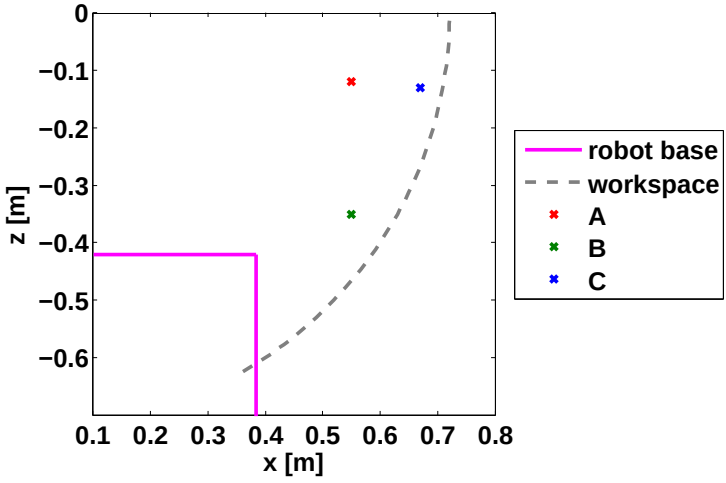


Figure 6.30: Location of the contact points in the robot workspace for experiments in the y-direction. The grey dashed line indicates the approximate boundary of the robot workspace.

Figure 6.31: Setup of the experiment where the PR2 robot has to apply a force on the force sensor (blue) attached on the side of a heavy structure. The tool attached on the force sensor allows applying a force in the y- and z-direction simultaneously.

6.7.2 Experimental determination of K_a

In a first set of experiments the robot makes contact with its gripper on the surface of a table, and is commanded to push on it with a desired force of $10N$. The robot applies force-sensorless force control in the Cartesian z-direction, while maintaining the orientation and planar position of its gripper. Each iteration of the experiment applies a different value of K_a. Figure 6.32 shows that **higher positive values increase damping, higher negative values lower damping**. This conforms to the theoretical analysis of Section 6.4.2 and 6.5.

Figure 6.33 shows the result of retaking the experiment, but increasing the force applied in the contact situation. The desired force is increased in steps of $5N$ every $5s$. By maintaining contact, the experiments avoid the impact of the gripper on the force sensor and resulting excitation of the force sensor dynamics. The figure shows that the low-level velocity gains are close to the optimal setting, with only a small overshoot. This overshoot can be damped with a small positive K_a.

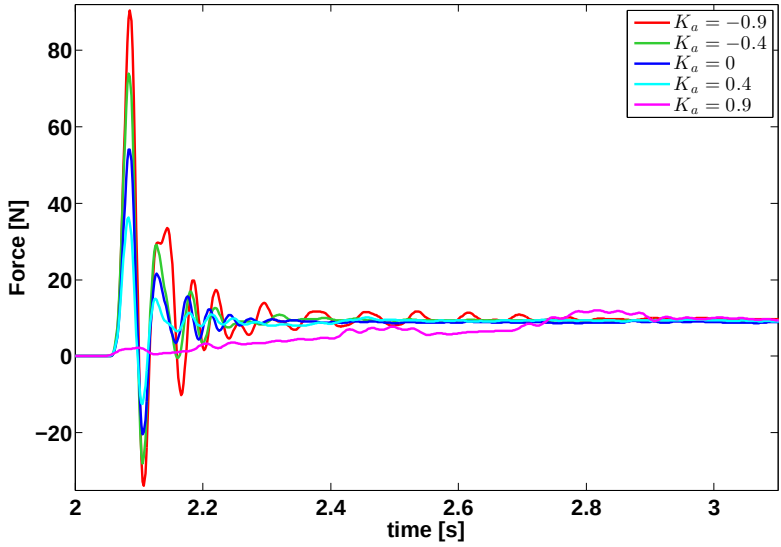


Figure 6.32: Transition from free space to contact ($\boldsymbol{w}_{d,sel} = F_{d,z} = 10N$) for different settings of K_a. Higher positive values increase damping, higher negative values lower damping. Excitation of the force sensor dynamics causes the negative forces on the figure.

Figure 6.34 shows the result of an even more refined set of K_a values. A K_a value of 0.1 is chosen, which approximates a critically damped step response.

6.7.3 Force constraint in a single direction

This section analyses the performance of the different control schemes introduced in Section 6.6 in different directions of a Cartesian task space. First experiments show the performance of the force control scheme which expresses the force task constraints in Cartesian task space for a desired force in the z-direction. Further experiments show the difference in performance with the force control scheme which expresses these force task constraints in joint task space. The last experiments of this section show the performance of the force control scheme which expresses the force task constraints in Cartesian task space for a desired force in the y-direction.

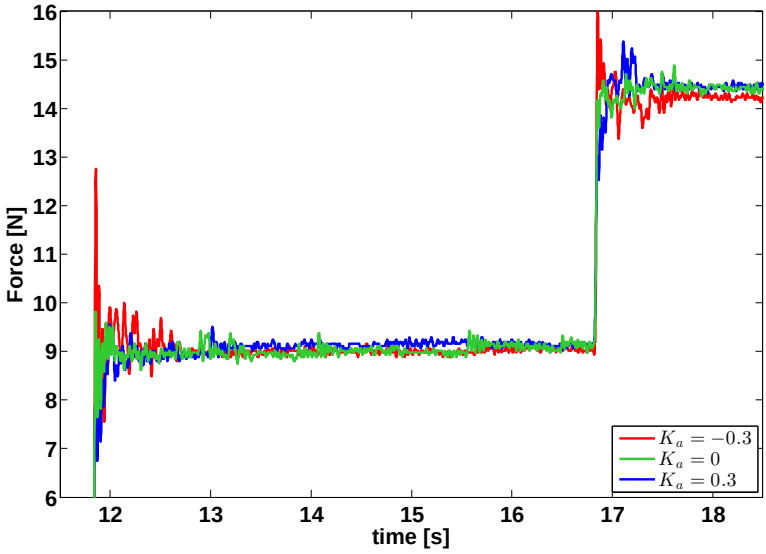


Figure 6.33: Transition to $\boldsymbol{w}_{d,sel} = F_{d,z} = 10N$ and $15N$ while maintaining contact, for different settings of K_a.

Force in z expressed as constraints in Cartesian task space

In this set of experiments the robot applies a force on a table with its gripper. The experiments use the control scheme described in Section 6.6.2, which expresses the force task constraints in Cartesian task space. A pose controller keeps the gripper vertical and on the contact point on the table. A joint configuration task is active as secondary constraint, keeping the robot elbow pointing outwards. Figure 6.35 shows the results of the experiments. The robot starts with its gripper in free space. A first setpoint of $5N$ moves the gripper down, until contact is made and a force built up. Every five seconds, the force is increased, first to $10N$, then $15N$; after which the force is decreased again, first to $10N$, then $5N$. The gripper maintains contact throughout the force build-up and reduction. The experiment is repeated ten times in each of the three contact points.

The initial contact with the force sensor on the table excites the force-sensor dynamics. As a result Figure 6.35 shows a larger peak in the transient for the first step in desired force than for the other steps. Further the figure shows that the transients damp out in less than half a second when building up the

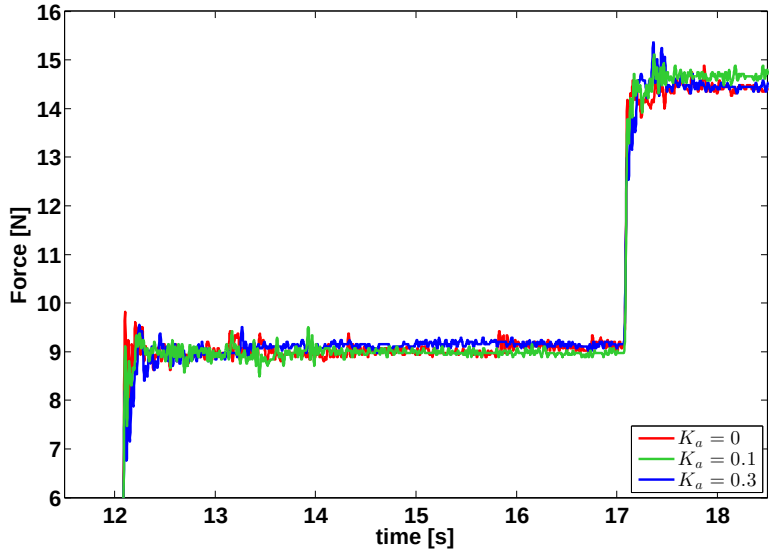


Figure 6.34: Transition to $\boldsymbol{w}_{d,sel} = F_{d,z} = 10N$ and $15N$ while maintaining contact, for different settings of K_a.

force. However, when reducing the applied force, the transients damp out in $1.5s$. After transients have damped out, the measurement converges to a constant force, characterized by its average and the standard deviation around this average.

F_d	5N	10N	15N	10N	5N
point A	0.09	0.10	0.11	0.13	0.10
point B	0.12	0.09	0.13	0.14	0.12
point C	0.06	0.06	0.08	0.10	0.09
total	0.09	0.08	0.11	0.12	0.10

Table 6.1: Average of the measurement standard deviations, expressed in Newton.

Table 6.1 shows the average of the measurement standard deviations over the ten repetitions of a measurement in a certain contact point. These average standard deviations for the different contact points and applied force magnitudes show little variation. Moreover, their values are low, about 1% of the applied force and only two to three times the sensor resolution. Therefore, we can conclude that a **stable, constant force is reached**.

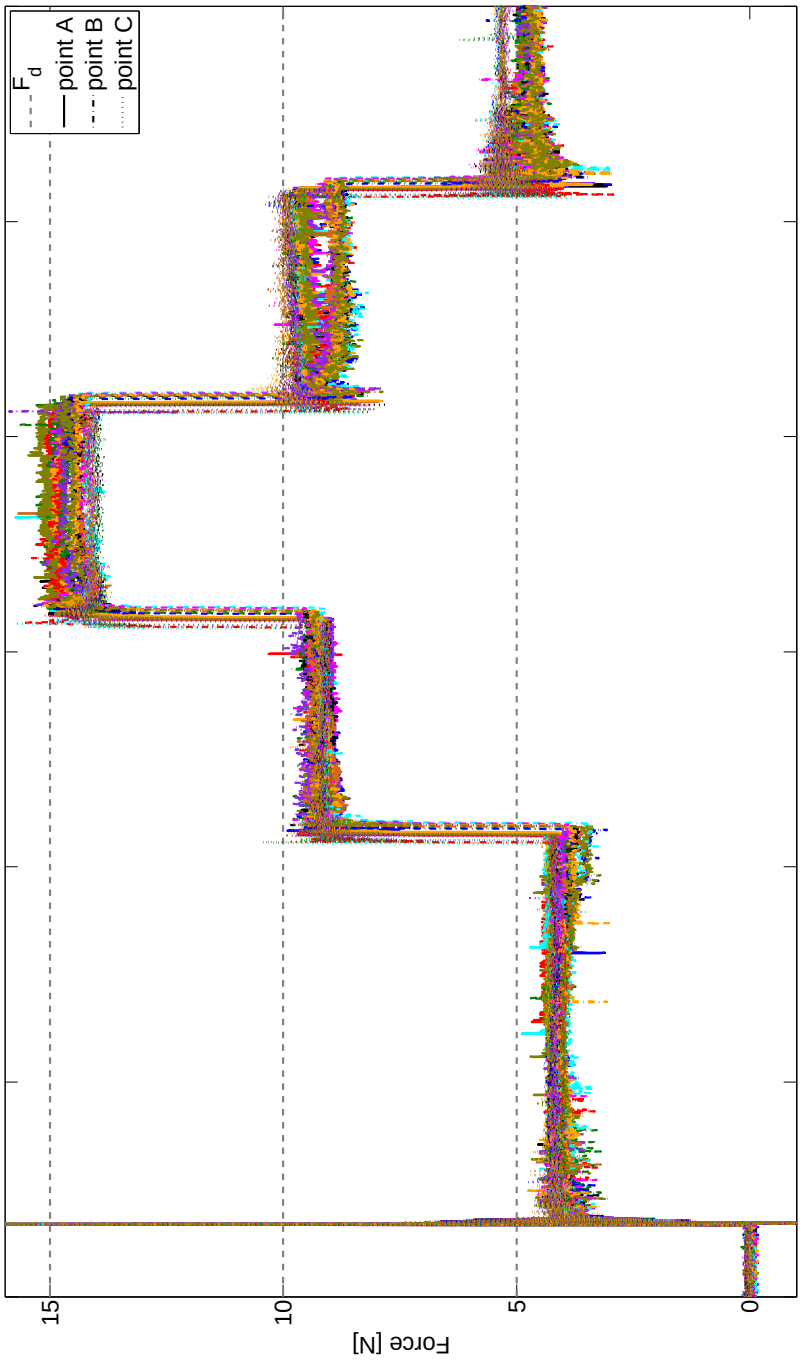


Figure 6.35: Measured force over time of the experiments applying a force in the z-direction with constraints expressed in Cartesian task space. Grey, dashed lines indicate the desired values. The experiments are repeated ten times, in three different contact points. Different line styles differentiate the contact points, different colors differentiate the experiments. The large spike shows the excitation of the force sensor dynamics when the gripper makes contact with it.

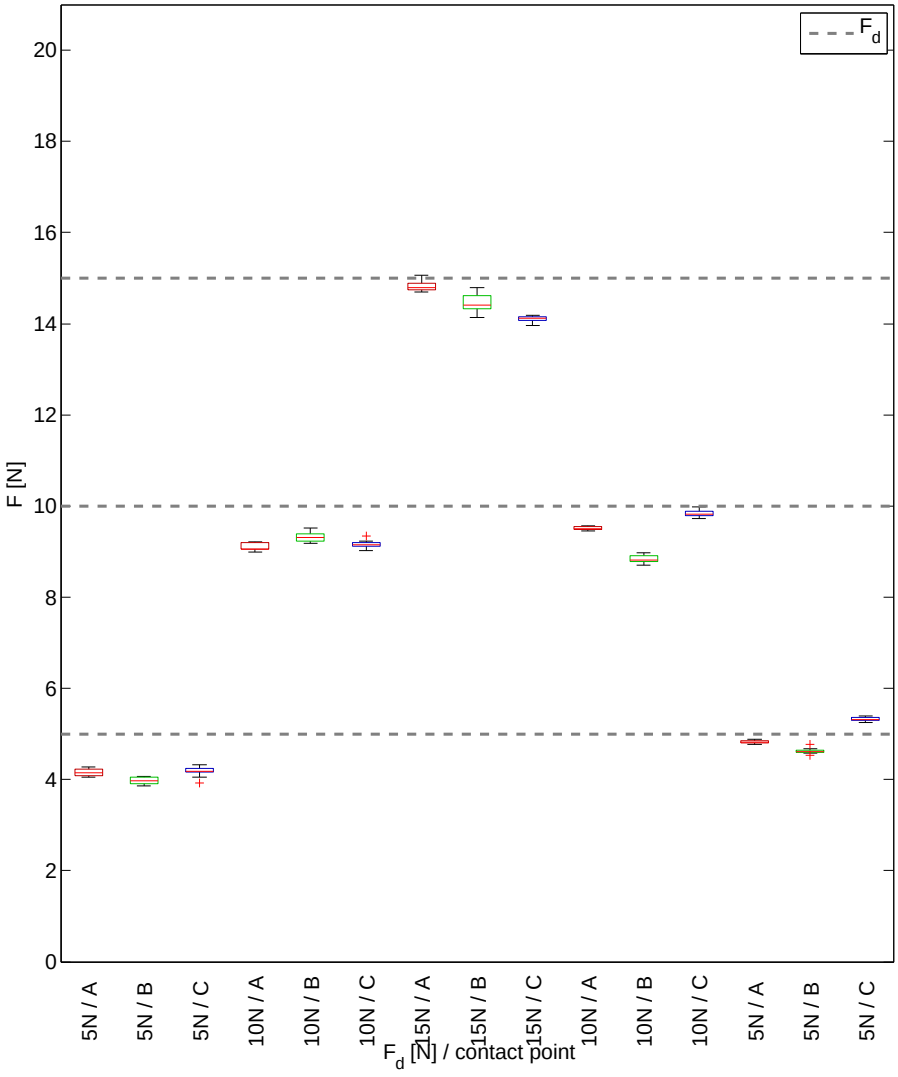


Figure 6.36: Boxplot of the steady-state averages of the experiments applying a force in the z-direction with constraints expressed in Cartesian task space. Grey, dashed lines indicate the desired values. The measured force over time is shown in Figure 6.35.

Figure 6.36 shows the distribution of the measurement averages for the different steps in applied force. It shows that the repeatability of the measurements in a certain contact point is high, with a standard deviation around $0.1N$. The distribution of the measurement averages between the different contact points is however larger than between the measurement repetitions in a certain contact point. This demonstrates the **robot configuration dependency of the accuracy**: different robot configurations result in the robot joints taking a different part of the load, as represented by the configuration dependency of the Jacobian matrix $\boldsymbol{J}_{\boldsymbol{q}}$. **The distribution between the contact points and also the accuracy of the measurements, are higher when decreasing the force than when building up the force.** In addition, remark that all averages of the measurements but one are lower than the expected values, as can expected since the control scheme does not take Coulomb friction into account. Appendix B.1 details the desired velocity $\dot{\boldsymbol{y}}_{d,1}^{\circ}$ over time.

Force in z expressed as constraints in joint task space

The experiments discussed in this paragraph repeats previous experiments, but replaces the force control scheme by the control scheme described in Section 6.6.3, which expresses the force control task constraints in joint task space. The force constraints span the whole robot joint space, hence no additional constraints are added.

Figure 6.37 shows the results of the experiments. The results are similar to the results shown in Figure 6.35, however they show lower accuracy when building up the force, and more dispersion between measurements at different contact points. Although the reference adaptation loop is different from the control scheme with constraints in Cartesian task space, the transient behavior is similar, as predicted by the theoretical analysis of Section 6.6.3.

F_d	5N	10N	15N	10N	5N
point A	0.09	0.09	0.10	0.05	0.07
point B	0.10	0.11	0.12	0.05	0.06
point C	0.07	0.11	0.05	0.05	0.09
total	0.09	0.10	0.09	0.05	0.07

Table 6.2: Average of the measurement standard deviations, expressed in Newton.

Table 6.2 shows the average of the measurement standard deviations over the ten repetitions of a measurement in a certain contact point. As for previous

experiments, these average standard deviations show little variation. Therefore, we can conclude that a stable, constant force is reached.

Figure 6.38 shows the distribution of the measurement averages for the different steps in applied force. It shows that, as in previous experiments, the repeatability of the measurements in a certain contact point is high, with a standard deviation around $0.1N$. Moreover, the distribution of the measurement averages between the different contact points shows the robot configuration dependency of the accuracy. However, in contrast to the experiments of the control scheme with constraints in Cartesian task space, this distribution is not different when building up or decreasing the force.

In contrast to the predictions of the theoretical analysis of Section 6.6.3, **the accuracy of the control scheme expressing the force control constraints in joint space is lower than the accuracy of the control scheme expressing these constraints in Cartesian space.** Possible explanations include: (i) First, the adapted control gains are large enough with respect to the robot damping, hence the projection error of the control scheme expressing the force control constraints in Cartesian space will be neglectable. (ii) Second, the difference between the results of both control schemes could be due to the robot configuration difference, which proves to be an important influence. This robot configuration determines next to the Jacobian matrix the division of the load between the different robot joints. The robot joints of the PR2 robot arm are however not similar, they have different levels of backdrivability and Coulomb friction. (iii) Third, in case of the control scheme expressing the force control constraints in joint space, the robot does not control the pose of the gripper. The optimization problem of this control scheme, i.e. pseudo-inverse of the augmented Jacobian $\boldsymbol{A}$, results in an (instantaneously) minimal weighted norm of the joint velocities. In practice, it is the shoulder joint of the robot arm that will contribute most to the approach of the robot arm to the table, resulting in a contact where the gripper makes an angle with the table. Therefore, the robot arm wrist joints will take a larger share of the load with respect to the control scheme expressing the force control constraints in Cartesian space. These wrist joints are known to be less back drivable than the other joints. Moreover, the pr2 robot wrist joints are not gravity compensated, hence the gripper will tend to point downwards. This effect is not accounted for in the control scheme, hence will disturb the applied joint torques.

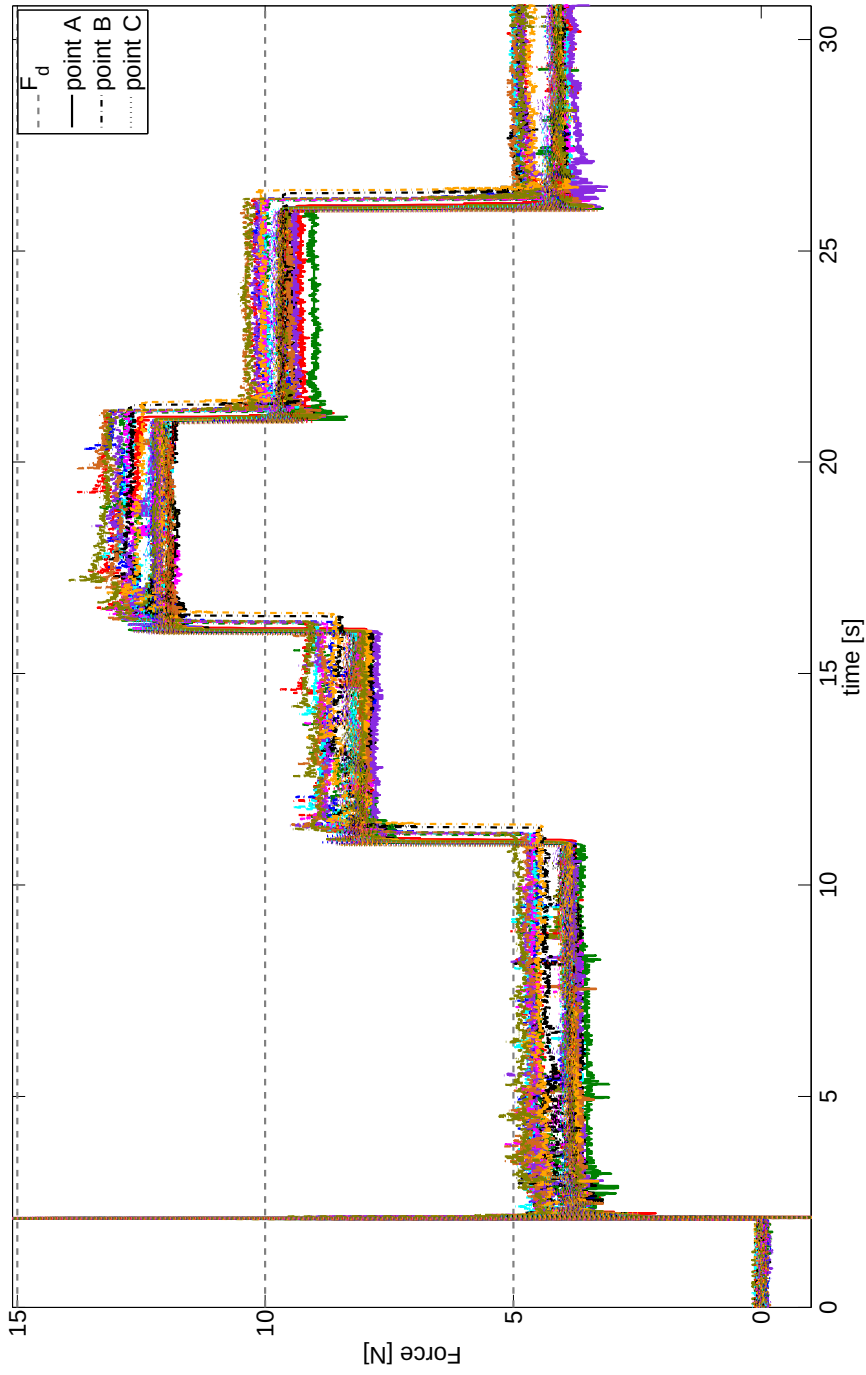


Figure 6.37: Measured force over time of the experiments applying a force in the z-direction with constraints expressed in joint task space. Grey, dashed lines indicate the desired values. The experiments are repeated ten times, in three different contact points. Different line styles differentiate the contact points, different colors differentiate experiments. The large spike shows the excitation of the force sensor dynamics when the gripper makes contact with it.

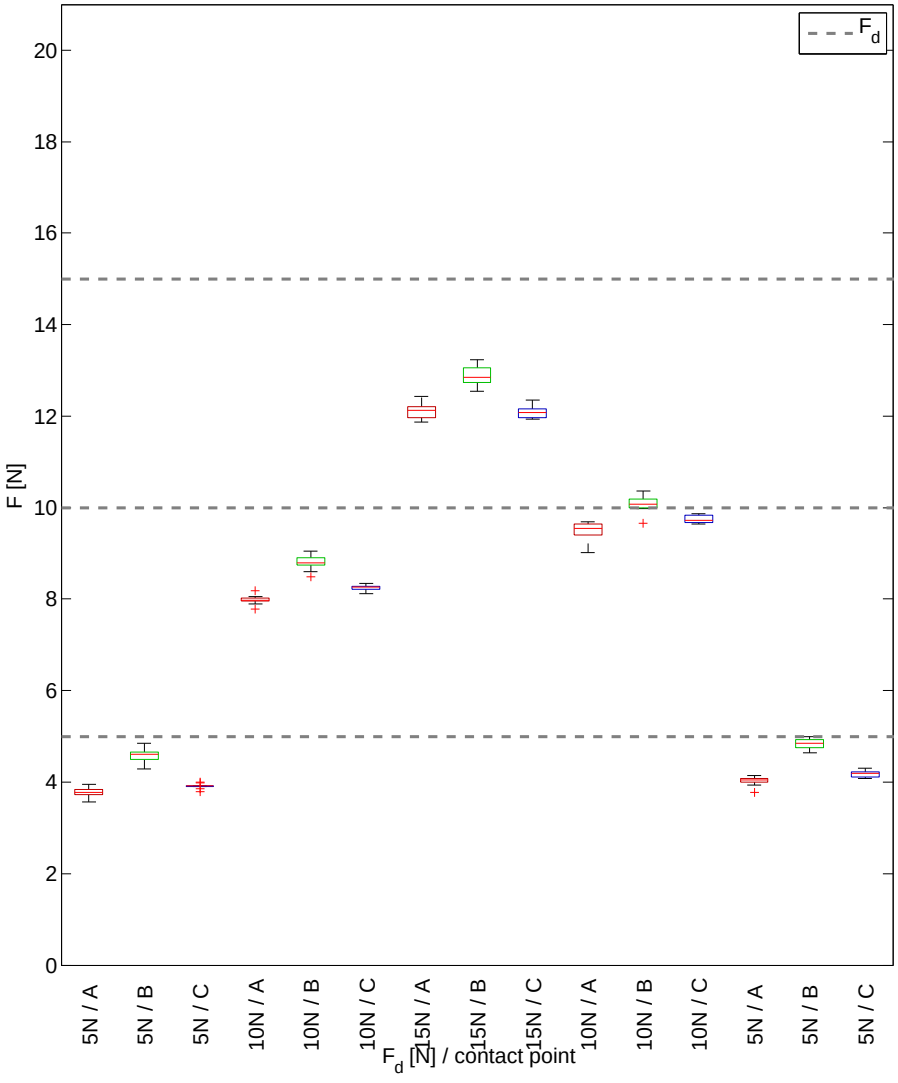


Figure 6.38: Boxplot of the steady-state averages of the experiments applying a force in the z-direction with constraints expressed in joint task space. Grey, dashed lines indicate the desired values. The measured force over time is shown in Figure 6.37.

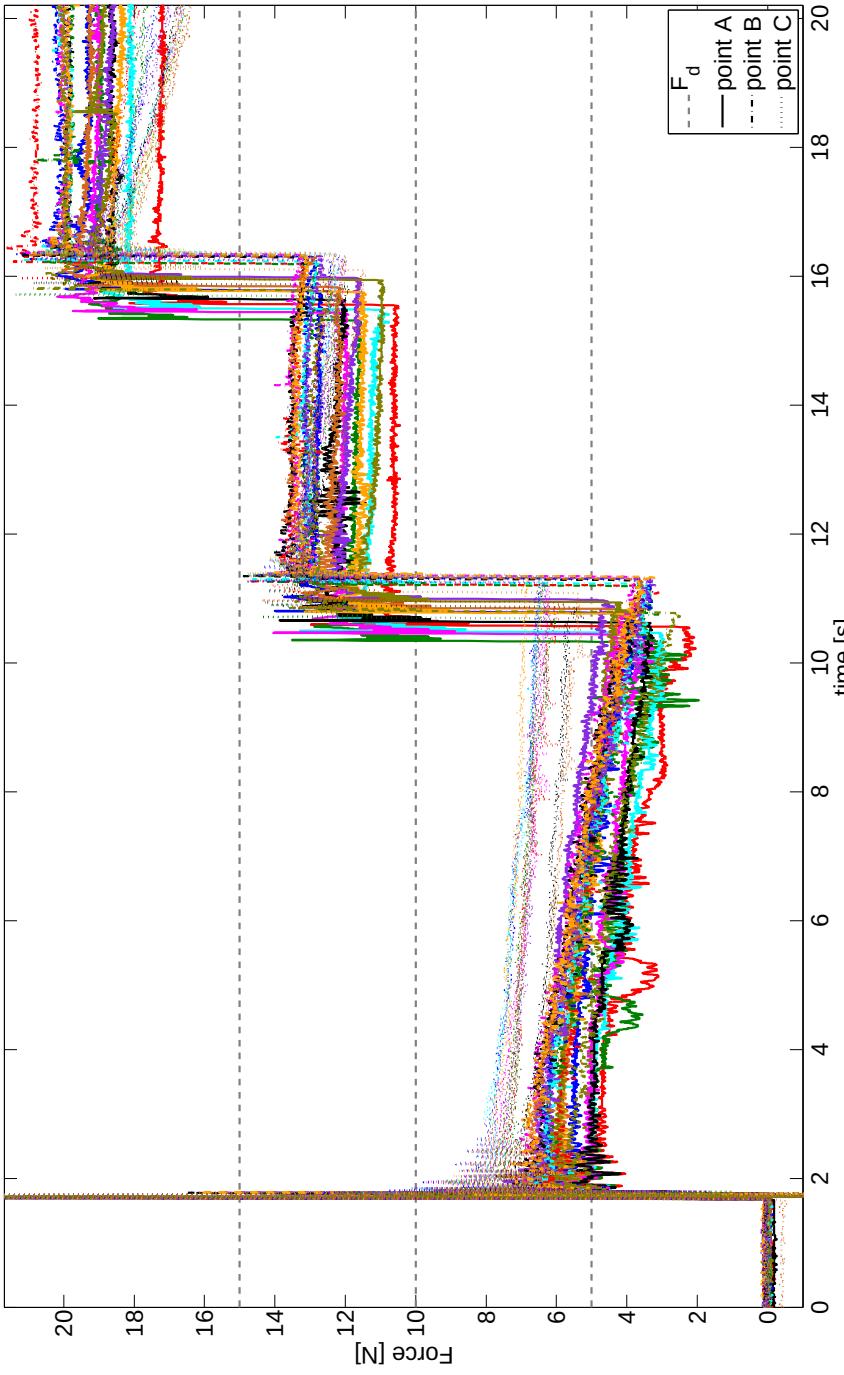


Figure 6.39: Measured force over time of the experiments applying a force in the y-direction with constraints expressed in Cartesian task space. Grey, dashed lines indicate the desired values. The experiments are repeated ten times, in three different contact points. Different line styles differentiate the contact points, different colors differentiate experiments. The large spike shows the excitation of the force sensor dynamics when the gripper makes contact with it.

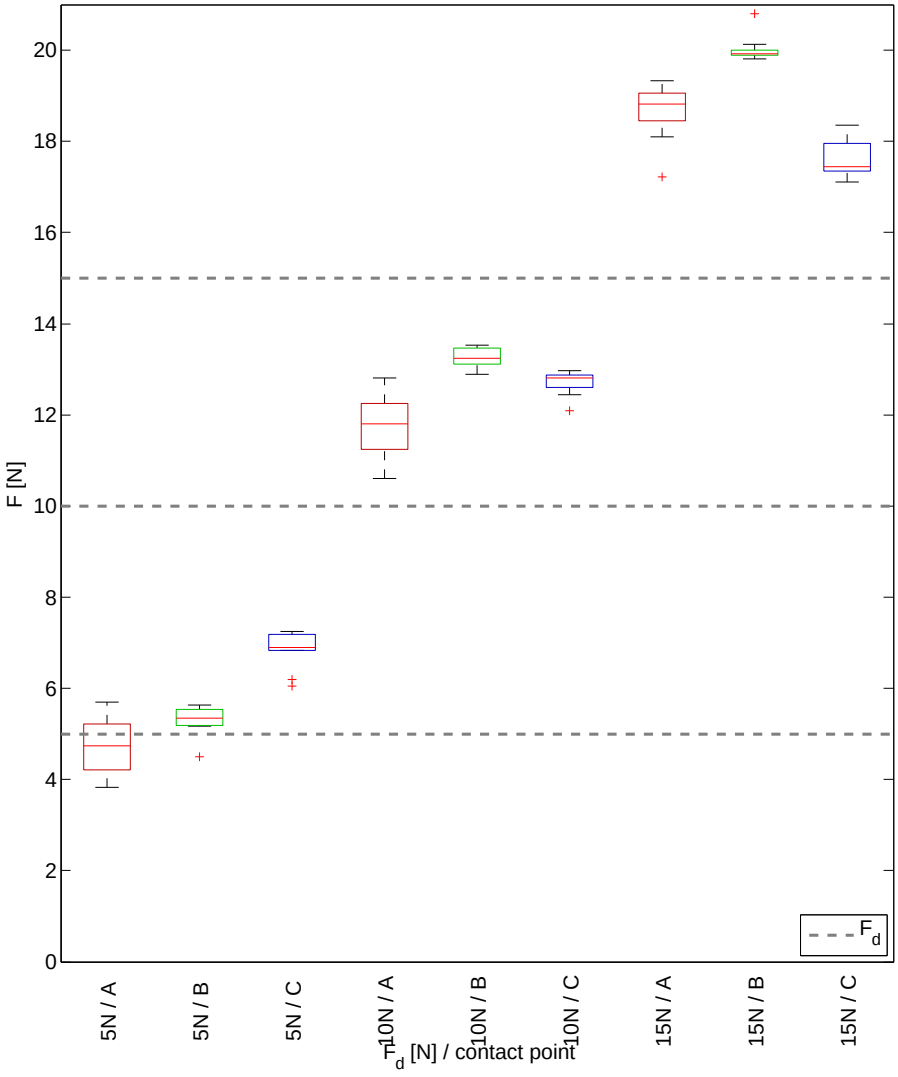


Figure 6.40: Boxplot of the steady-state averages of the experiments applying a force in the y-direction with constraints expressed in Cartesian task space. Grey, dashed lines indicate the desired values. The measured force over time is shown in Figure 6.39.

Force in y expressed as constraints in Cartesian task space

The experiments discussed in this paragraph considers a similar setup as for the experiments applying a force on a table using the control scheme that expresses the force control constraints in Cartesian task space. However, in the experiments discussed in this paragraph, the robot applies a force on the side of a heavy structure instead of a table. The pose controller keeps the gripper horizontal and on the contact point on the structure.

Figure 6.39 shows the results of the experiments. In contrast to previous experiments, the force applied on the environment is higher than the desired value. Moreover, the applied force tends to drift a little over time. The response shows a little larger overshoot with respect to previous experiments. The reference adaptation factor K_a is tuned for the z-direction, which could lead to different results in the y-direction, and hence could explain the little larger overshoot.

F_d	5N	10N	15N
point A	0.46	0.11	0.09
point B	0.62	0.09	0.08
point C	0.29	0.24	0.42
total	0.46	0.15	0.20

Table 6.3: Average of the measurement standard deviations, expressed in Newton.

Table 6.3 shows the average of the measurement standard deviations over the ten repetitions of a measurement in a certain contact point. These average standard deviations show a similar repeatability as previous experiments, with exception for the experiments that drift, i.e. in contact point C and for the initial force of $5N$.

Figure 6.40 shows the distribution of the measurement averages for the different steps in applied force. It shows a lower repeatability of the measurements in a certain contact point with respect to previous experiments, with a dispersion four to six times as high. Moreover, the dispersion between measurements at different contact points is about three times as high. The experiments demonstrate the **lowest accuracy of all experiments**. The experiment with the largest difference between its average and the desired value is $5.8N$, for a desired value of $15N$. Figure 6.39 shows this experiment in red dash-dotted line.

A possible explanation of the results is, next to configuration dependency, that the pr2 robot **wrist joints are not gravity compensated, hence the**

gripper will tend to point downwards. This effect is not compensated for in the control scheme, hence the applied force will differ from the desired one. In contrast, this will have no effect when applying a force in the z-direction with the gripper already pointing downward.

There is no data when reducing the desired force, hence no conclusions can be made whether decreasing the force results in better accuracy. However, next subsection will detail the combination of a force in the y-direction with a force in the z-direction, hinting at a better accuracy when decreasing the desired force.

6.7.4 Force constraints in multiple directions

This section analyses the performance of the force control scheme which expresses the force control task constraints in Cartesian task space when applying a force simultaneously in different directions of a Cartesian task space. First experiments show the performance when applying a desired force in the z-direction, explicitly specifying the force in other directions as zero. Further experiments show the performance when applying a desired force in the Cartesian y- and z-direction.

Force in the z-direction, explicitly specifying the force in other directions as zero

The experiments discussed in this paragraph retakes the experimental setup where the robot has to apply a force on a table using the control scheme that expresses the force control constraints in Cartesian task space. However, in this set of experiments there is no pose control task; the Cartesian directions other than the z-direction are explicitly constraint to a force or torque of zero N or Nm. The gripper will start in a vertical pose, but no controller will enforce this pose. In practice, however, there is little deviation from the vertical pose.

Figure 6.41 shows the results of the experiments. The results are similar to the result shown in Figure 6.35, however show little lower accuracy when building up the force. The figure shows a longer time before the measurements convergence to a constant value, most notably for a desired force of $15N$ and the desired force reduced to $5N$. **Since no controller enforces position constraints, the robot arm moves the contact point when the force setpoint is altered.** In case of the 15N and the decreased 5N desired force, the robot moves the arm and gripper horizontally before reaching equilibrium, as can be seen on the video [164] explained in Appendix C.3.4. This movement results in the

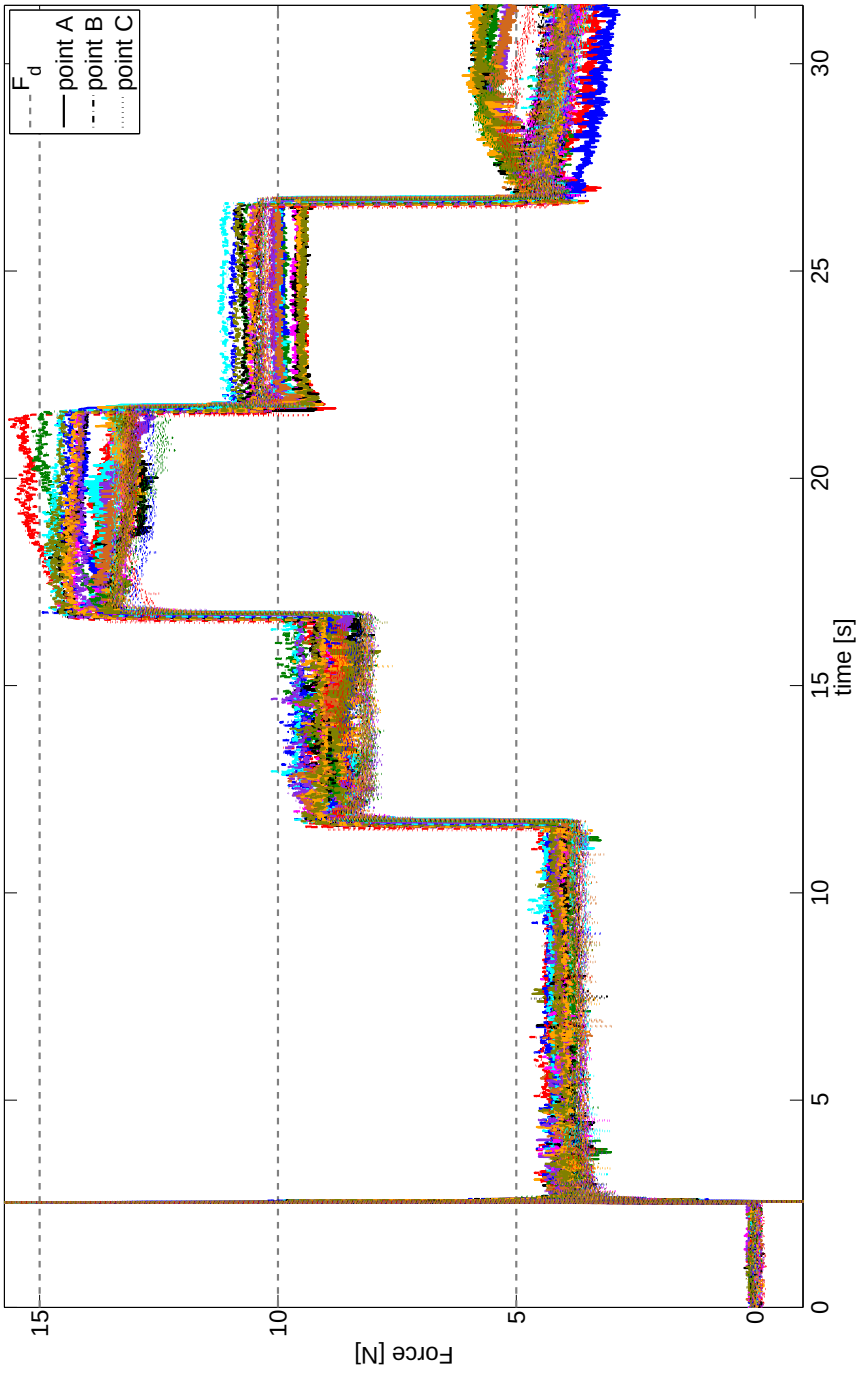


Figure 6.41: Measured force over time of the experiments applying a force [0 0 x 0 0 0] with constraints expressed in Cartesian task space. Grey, dashed lines indicate the desired values. The experiments are repeated ten times, in three different contact points. Different line styles differentiate the contact points, different colors differentiate experiments. The large spike shows the excitation of the force sensor dynamics when the gripper makes contact with it.

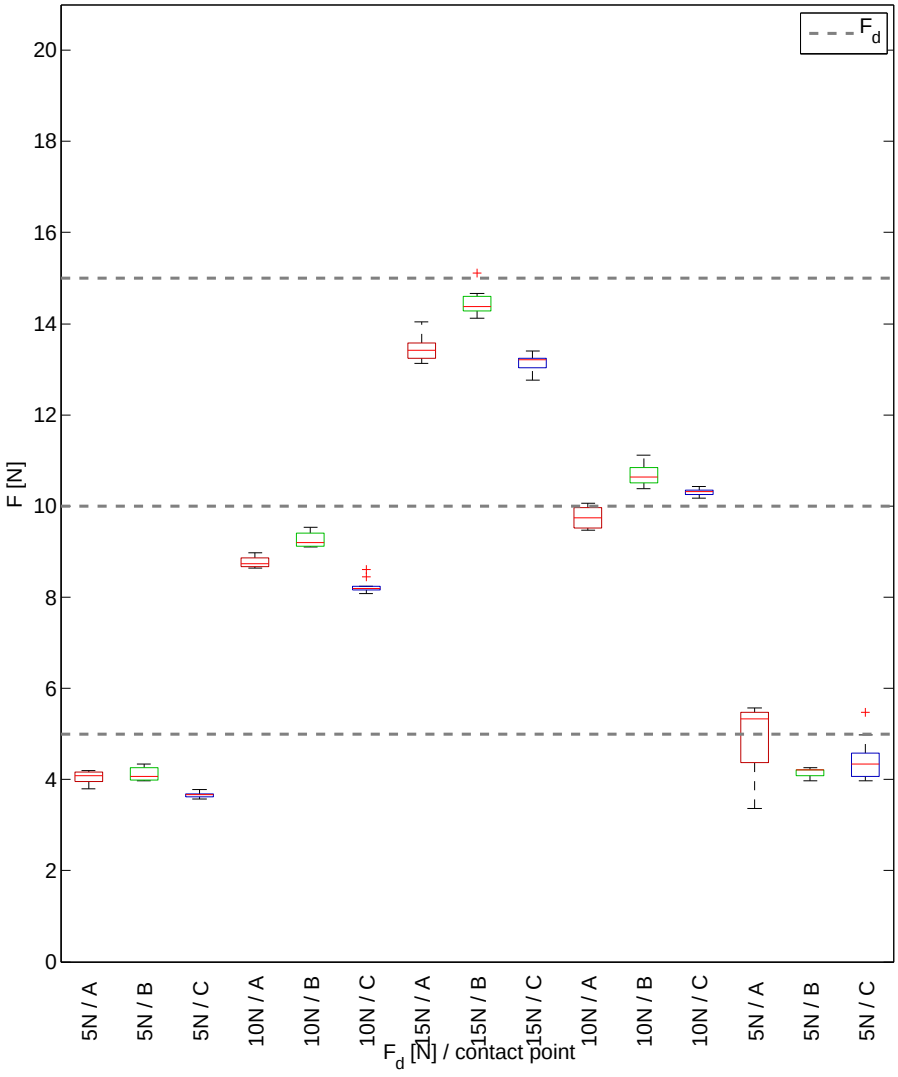


Figure 6.42: Boxplot of the steady-state averages of the experiments applying a force $[0\ 0\ x\ 0\ 0\ 0]$ with constraints expressed in Cartesian task space. Grey, dashed lines indicate the desired values. The measured force over time is shown in Figure 6.41.

slower build-up of the force, hence the slower convergence. Moreover, the contact point changes considerably, hence the robot resides in a different configuration. This new configuration adds to the divergence of the measurement results. Table 6.4 shows the average of the measurement standard deviations over the ten repetitions of a measurement in a certain contact point. These average standard deviations show little variation, as for previous experiments, with exception of the experiments where there is a slow build-up of the force due to the arm movement.

F_d	5N	10N	15N	10N	5N
point A	0.08	0.20	0.19	0.06	0.31
point B	0.11	0.17	0.14	0.06	0.20
point C	0.09	0.12	0.13	0.05	0.24
total	0.10	0.16	0.16	0.06	0.25

Table 6.4: Average of the measurement standard deviations, expressed in Newton.

Figure 6.38 shows the distribution of the measurement averages for the different steps in applied force. It shows that, as in previous experiments, the repeatability of the measurements in a certain contact point is high, with exception of the experiments where there is a slow build-up of the force due to the arm movement. As in previous experiments, the figure shows a better accuracy when decreasing the force than when building the force up. **Using the full Cartesian task space to apply a wrench conforming to a force vector in the z-direction, does not show improved performance with respect to only selecting the z-direction to apply the force.**

For most use cases it is of interest to control the contact point, hence the possible arm movement when changing the desired force will be undesirable. As a consequence this scheme offers no advantages with respect to the control scheme that expresses the force control constraints in Cartesian task space for use cases such as applying a force on a table.

Appendix B.1 details the desired velocity $\dot{\boldsymbol{y}}_{d,1}^{\circ}$ over time.

Applying a force in the Cartesian y- and z-direction

In the experiments discussed in this paragraph, the PR2 robot applies a force in the Cartesian y- and z-direction simultaneously. The robot applies these forces on an object attached to a force sensor, shown in Figure 6.31. A pose controller enforces a fixed orientation and x-position of the gripper.

Figures 6.43 and 6.45 show the measured force over time of the experiments in the y- and z-direction respectively. Both show **drift on the measurements in both directions**, as for some of the experiments in the single y-direction. This drift is also noticeable in the measurement standard deviations, shown in tables 6.5 and 6.6. These tables show the average of the measurement standard deviations over the ten repetitions of a measurement in a certain contact point.

F_d	5N	10N	15N	10N	5N
point A	0.55	0.34	0.30	0.22	0.28
point B	0.90	0.38	0.12	0.36	0.59
point C	0.44	0.23	0.12	0.17	0.18
total	0.63	0.32	0.18	0.25	0.35

Table 6.5: Average of the measurement standard deviations, expressed in Newton.

F_d	5N	10N	15N	10N	5N
point A	0.21	0.15	0.30	0.16	0.14
point B	0.34	0.46	0.24	0.22	0.15
point C	0.18	0.19	0.23	0.13	0.19
total	0.24	0.27	0.26	0.17	0.16

Table 6.6: Average of the measurement standard deviations, expressed in Newton.

Figures 6.46 and 6.46 show the distribution of the measurement averages for the different steps in applied force. It shows that the repeatability of the measurements in a certain contact point remains high in the z-direction while it degrades in the y-direction. As for all previous experiments, the distribution of the measurement averages between the different contact points is larger than between the measurement repetitions in a certain contact point. The measurements are higher than the desired value in both directions when building up the force.

Figure 6.46 of the y-direction shows a very similar performance to the experiments in the y-direction shown in Figure 6.40. Therefore, **the conclusions of the y-direction experiments apply here**, most importantly the lack of gripper gravity compensation. This non-compensated weight could be an explanation for the higher measured forces in the z-direction with respect to other experiments in the z-direction. Moreover, the drift in the y-direction will cause an effect in the z-direction.

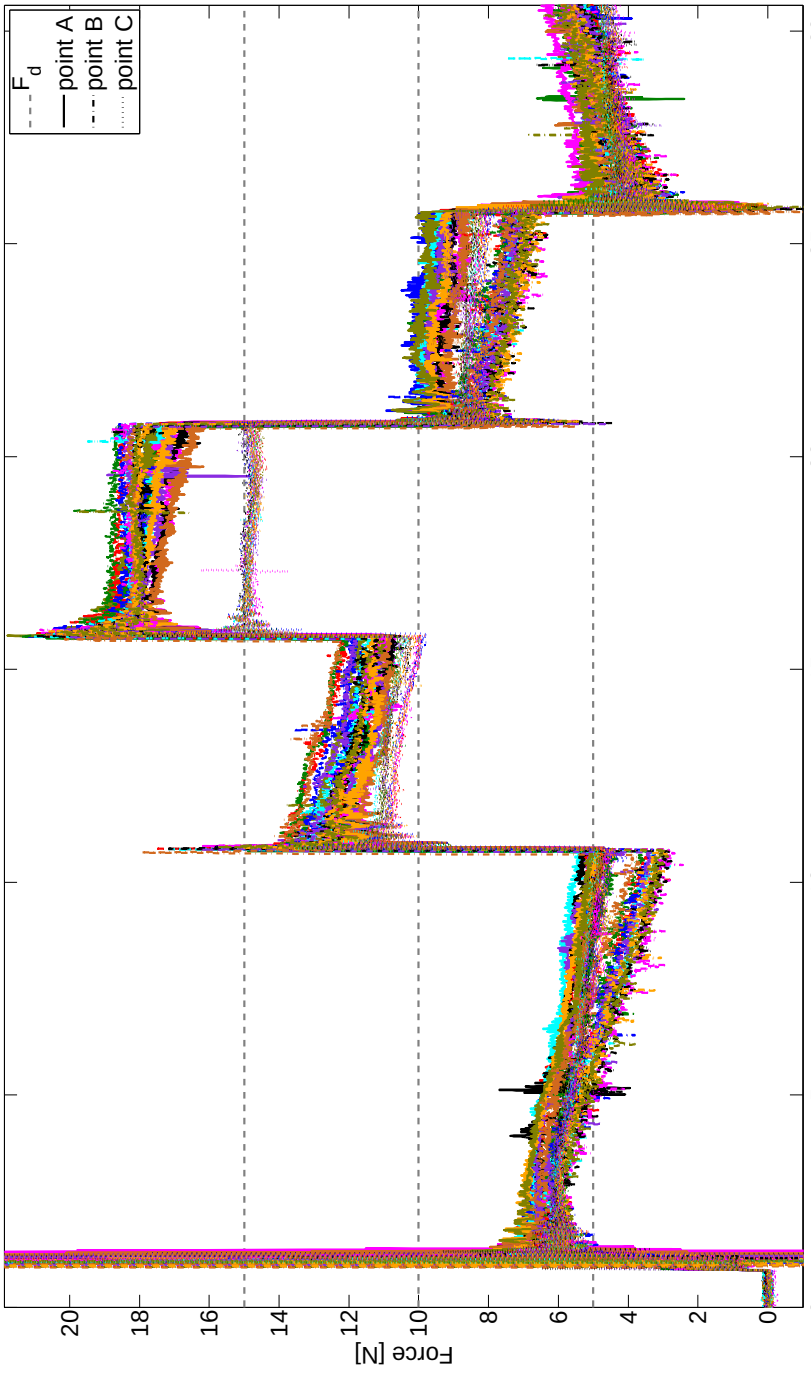


Figure 6.43: Measured force over time of the y-direction of the experiments applying a force in the yz-direction with constraints expressed in Cartesian task space. Grey, dashed lines indicate the desired values. The experiments are repeated ten times, in three different contact points. Different line styles differentiate the contact points, different colors differentiate experiments. The large spike shows the excitation of the force sensor dynamics when the gripper makes contact with it.

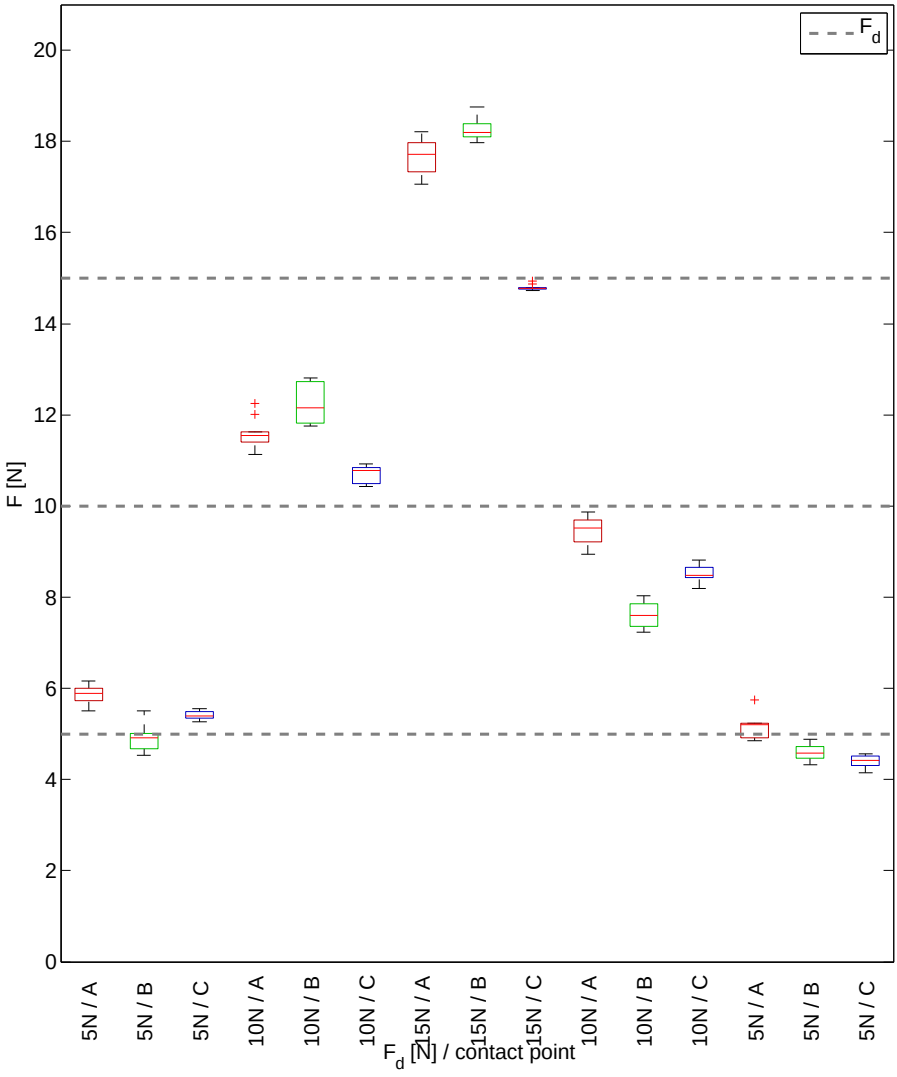


Figure 6.44: Boxplot of the steady-state averages of the y-direction of the experiments applying a force in the yz-direction with constraints expressed in Cartesian task space. Grey, dashed lines indicate the desired values. The measured force over time is shown in Figure 6.43.

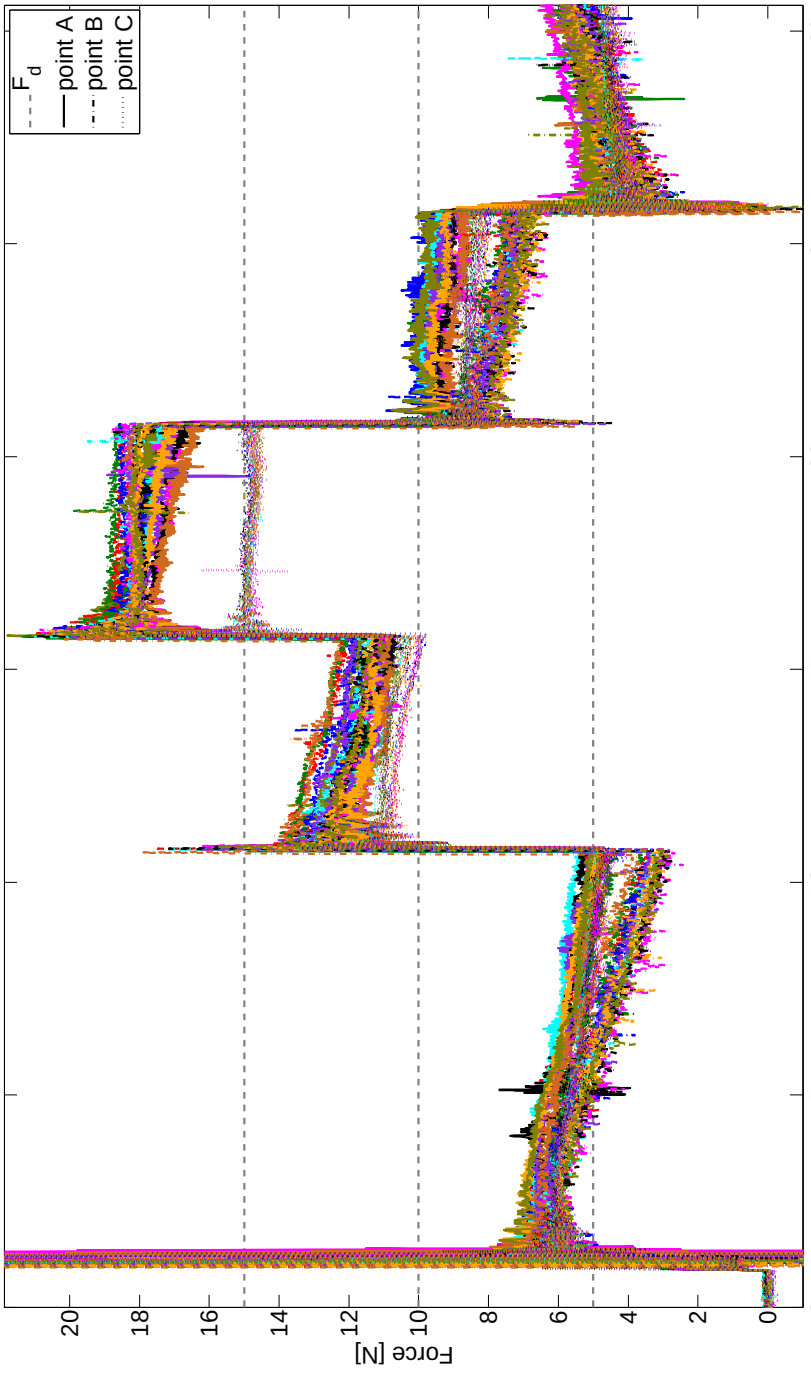


Figure 6.45: Measured force over time of the z-direction of the experiments applying a force in the yz-direction with constraints expressed in Cartesian task space. Grey, dashed lines indicate the desired values. The experiments are repeated ten times, in three different contact points. Different line styles differentiate the contact points, different colors differentiate experiments. The large spike shows the excitation of the force sensor dynamics when the gripper makes contact with it.

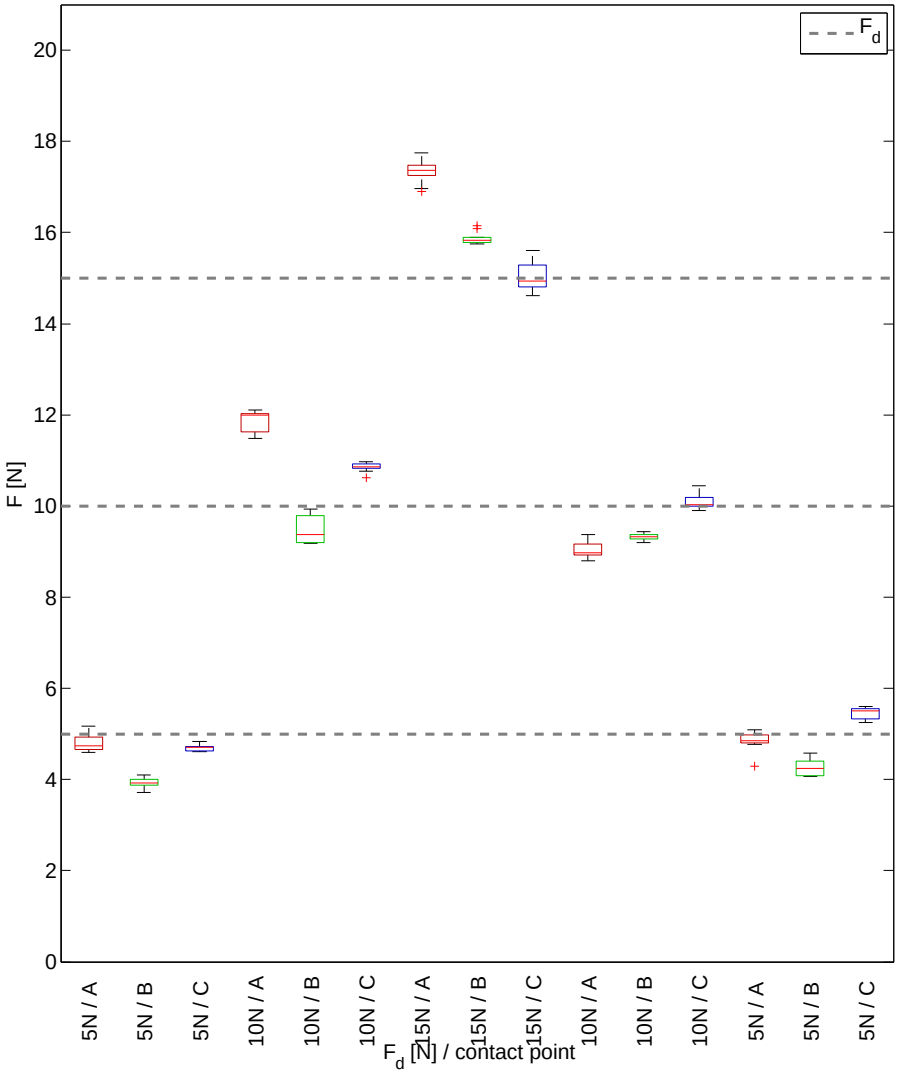


Figure 6.46: Boxplot of the steady-state averages of the z-direction of the experiments applying a force in the yz-direction with constraints expressed in Cartesian task space. Grey, dashed lines indicate the desired values. The measured force over time is shown in Figure 6.45.

Appendix B.1 details the desired velocity $\dot{\boldsymbol{y}}_{d,1}^{\circ}$ over time.

6.7.5 Table wiping use case

This section analyses the table wiping use case introduced in Section 6.6. Figure 6.47 shows the experiment set-up. The use case consists of a sequence of two sets of tasks, (i) the first set of tasks equals to the experiment where a force is applied on a table with the force control task constraints expressed in Cartesian task space, (ii) the second set of tasks equals the first set, but replaces the fixed position constraint in the x- and y-directions (the plane of the table) with a trajectory tracing a Lissajous figure, i.e. the *wiping movement*. Since the robot arm is moving, the steady-state condition for the application of the control schemes presented in Section 6.6 does not hold for the robot joints.

The experiment is repeated around two different points on the table, D and E shown in Figure 6.29. Around these points the robot repeats the experiment, tracing two different Lissajous figures at two different speeds.

This section focuses on one of the Lissajous figures, the circle, and one point, point D. Appendix B.4 details the other experiments.

Figures 6.48 and 6.49 show the circles traced by the robot gripper on the table for the two different tracing speeds. The textile of the wiper has non-negligible **friction** on the table, hence the imperfect tracking of the circle. The tracking performance is however of no concern to the use case and hence no focus of the experiment. Remark that the wiper and its mount changes the dynamics of the robot arm, however this does not alters the control strategy. However, the robot kinematics takes the wiper into account. Since the wiper deforms under higher forces, the exact contact point can differ slightly from the kinematic model.

Figures 6.50 and 6.51 show the force applied on the table F and the desired control task velocity $\dot{\boldsymbol{y}}_{ex}$ both as a function of time t. During the first five to ten seconds the robot gripper makes contact with the table and applies a force on the table. The contact point will be the center of the circle to trace on the table. After these five seconds, the gripper moves from the centre of the circle to the circle and starts wiping the table in a circular pattern.

The sensor is nulled between experiments. However the experimental setup can only measure relative errors, since the measurement is disturbed by the sensor and wiper mount. Therefore, the force averages are offset with respect to the real applied average force and can only be compared relatively. Figures 6.50 and 6.51 show the initial applied force aligned with the desired force $F_d = 10N$, in order to put the variation in perspective to the desired force. Therefore, the initial

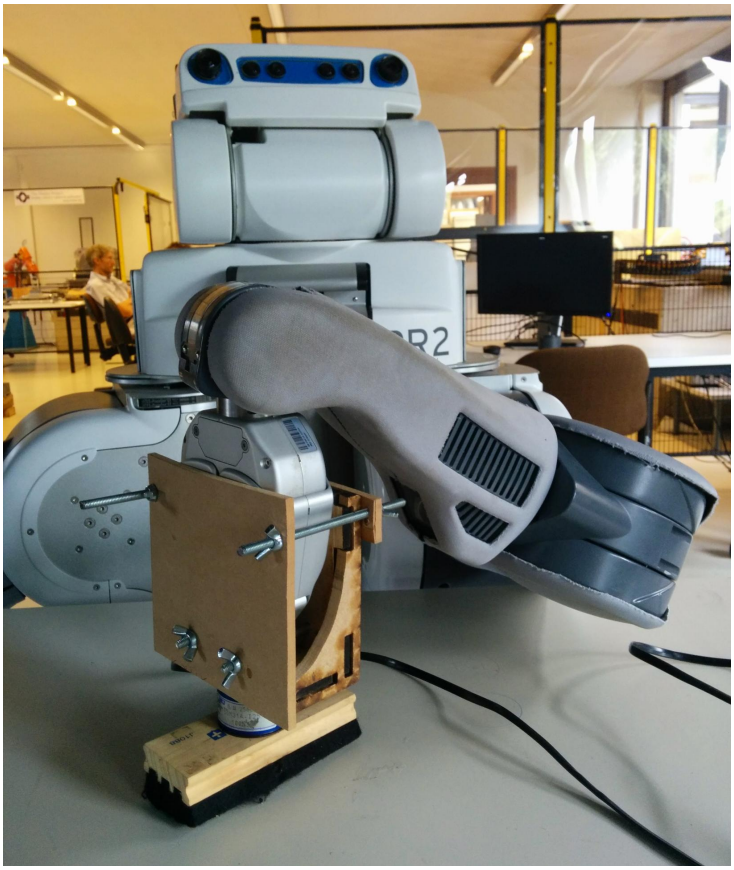


Figure 6.47: Table wiping use case setup. A force sensor is mounted between the gripper and the wiper. This sensor is only used for validation; it is not used in the control loop.

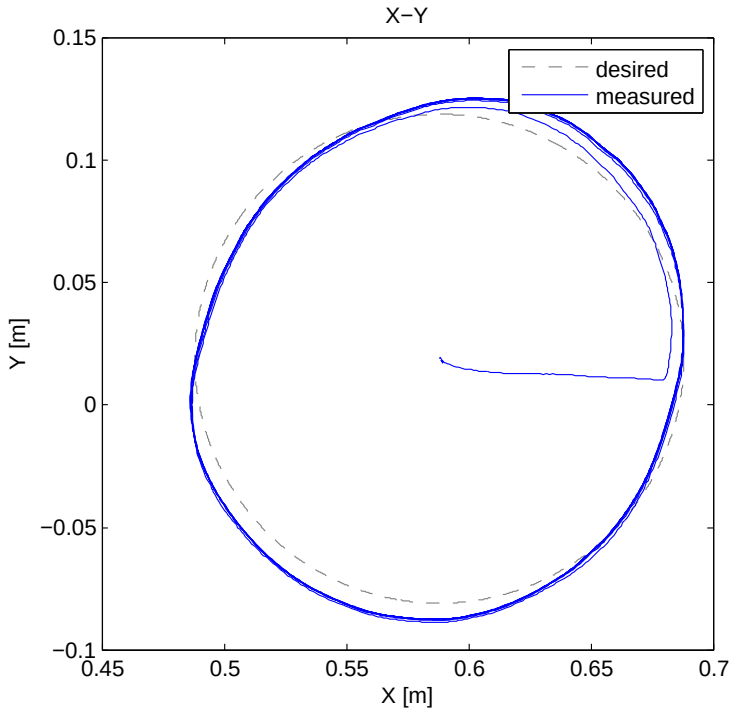


Figure 6.48: Circle traced by the robot gripper on the table around point D with a period of 20s.

steady-state error should be substracted from this value. Section 6.7.3, which analyses the experiments where a force is applied on a table with the force constraints expressed in Cartesian task space, shows that this **steady-state error is on average 0.8N and in worst-case 1.02N for a desired force F_d of $10N$. Therefore, the worst case deviation from the desired force setpoint during wiping is about 3N.**

The measured signal contains low-frequent, periodic oscillations. The period of these oscillations equals the period of the traced Lissajous figure. As a result, tracing the circle faster results in the same oscillation, but faster. This result suggests that **the oscillations of the measured force are mainly position and hence robot configuration dependent.** The measured signal also contains high-frequent oscillations, noise. One of the sources is the **stick-slip effect**, as can be seen in the videos [164] explained in Appendix C.3.5.

The oscillation does not lead to loss of contact or out-of-bound motor torques, hence the presented control scheme forms an acceptable

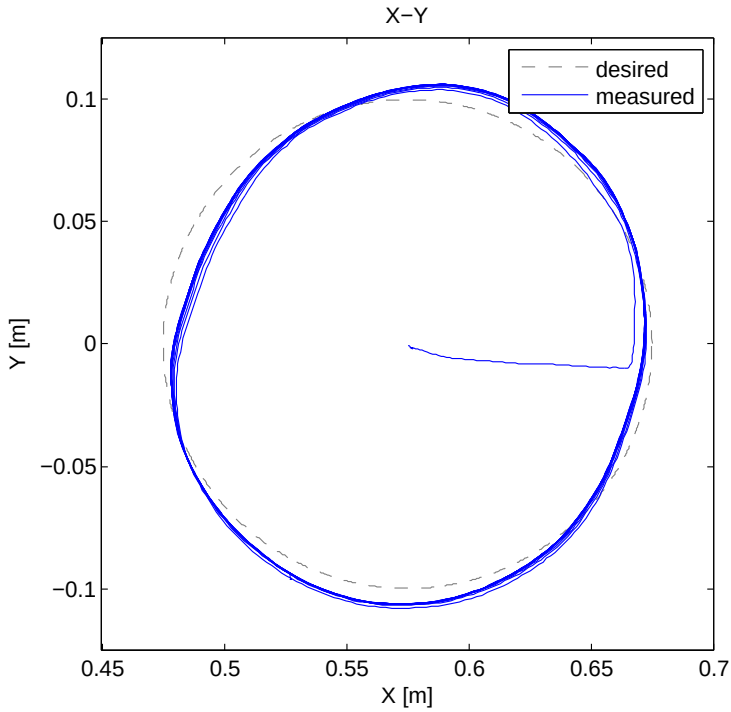


Figure 6.49: Circle traced by the robot gripper on the table around point D with a period of 10s.

control scheme for the analysed use case of table wiping.

6.7.6 Discussion and conclusions

This section first validated the effect of the reference adaptation factor K_a on the force response damping. A value of 0.1 for K_a was chosen and used throughout the experiments.

The experiments show that the control schemes proposed in previous section result in a stable, constant contact force. The experiments are highly repeatable in a certain contact point, with an average standard deviation around $0.1N$. However, the accuracy of the applied force with respect to the desired force depends on the robot configuration.

Figure 6.52 gives an overview of the averages of the experiments analysed in this section. The left side of the figure shows a linear force build-up when increasing

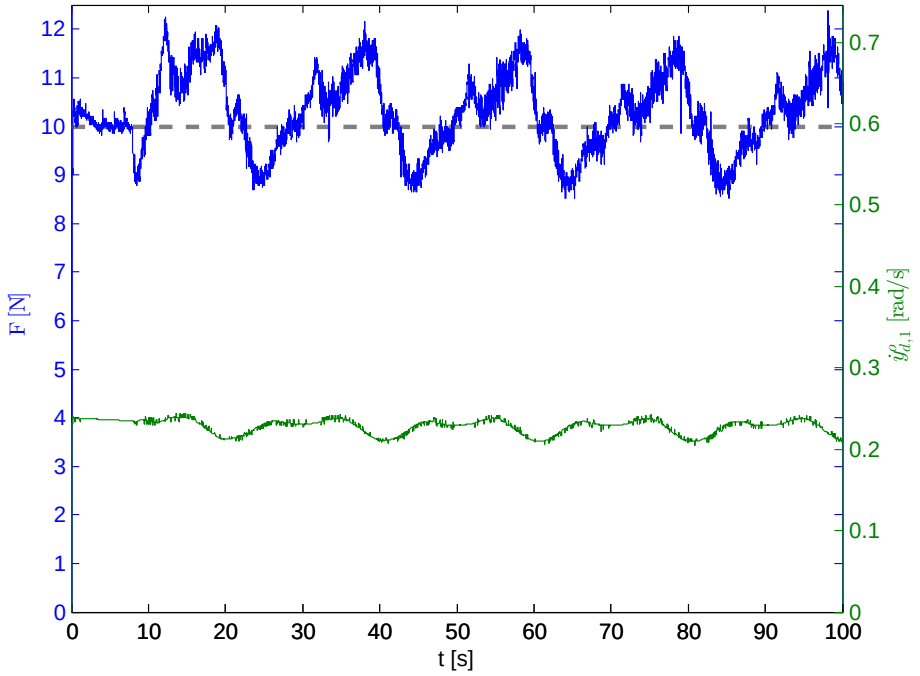


Figure 6.50: Force applied on the table F and desired control task velocity $\dot{\boldsymbol{y}}_{ex}$ over time t when tracing a circle around point D with a period of 20s. A grey dashed line indicates the desired force F_d to be applied on the table of 10N.

the desired force. Further, it shows that measurements with a component in the y-direction all have averages above the desired value. Applying a force in the y-direction shows also lower accuracy than the z-direction and sometimes drift. A possible explanation is the non-gravity compensated wrist joints of the PR2 robot, causing the gripper to point downwards. This gripper-down configuration is desired when applying a force in the z-direction, and hence does not disturb the controller. However, this gripper configuration is not desirable when applying forces in other directions, where it will disturb the controller since the gravity of the gripper is not modeled. Including gripper weight compensation on joint level could possibly increase accuracy. In contrast, the averages of the experiments in the z-direction are all lower than the desired value. These on average lower measured forces are expected since there are no disturbances such as Coulomb friction taken into account in the feedforward calculation.

The right side of Figure 6.52 shows that the averages of all experiments have

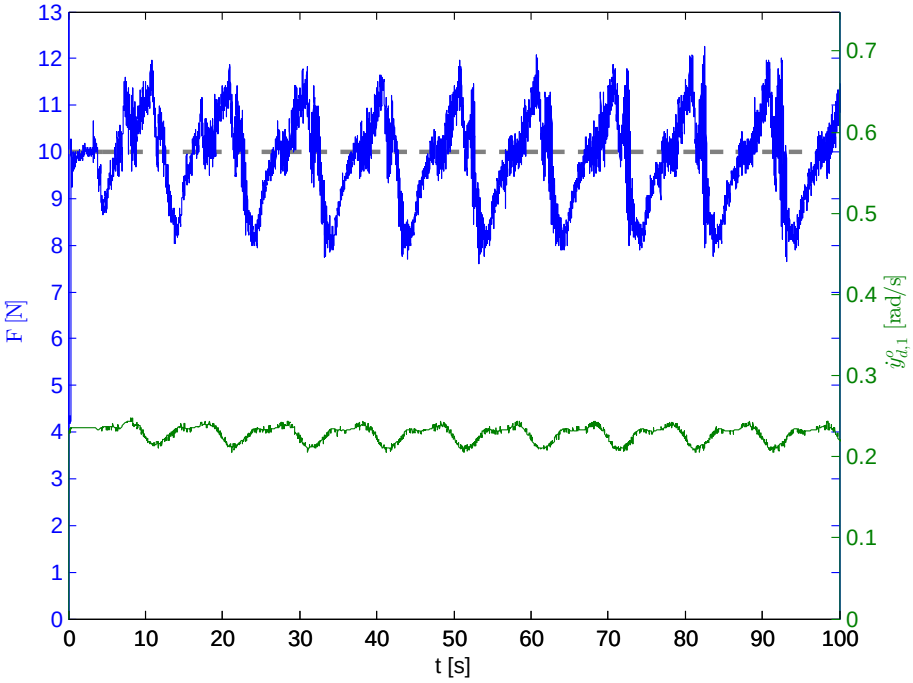


Figure 6.51: Force applied on the table F and desired control task velocity $\dot{\boldsymbol{y}}_{ex}$ over time t when tracing a circle around point D with a period of 10s. A grey dashed line indicates the desired force F_d to be applied on the table of 10N.

a higher accuracy when decreasing the desired force than when increasing it. This effect hints at hysteresis in the friction, possibly caused by the passive spring counterbalance system of the robot arm. This system compensates the weight of the arm. Future work should analyse this effect.

Figure 6.52 also confirms that the force control scheme of Section 6.6.3, which expresses the force control task constraints in joint space, does not provide a better performance than the force control scheme of Section 6.6.2, which expresses the force control task constraints in Cartesian space.

Using the full Cartesian task space to apply a wrench conforming to a force vector in the z-direction does not show improved performance with respect to selecting only the z-direction to apply the force. Since this task spans the whole Cartesian task space, only conflicting pose control task can be added. Therefore, the contact point cannot be controlled, and can change when altering the desired force. However, for most use cases it is of interest to control the

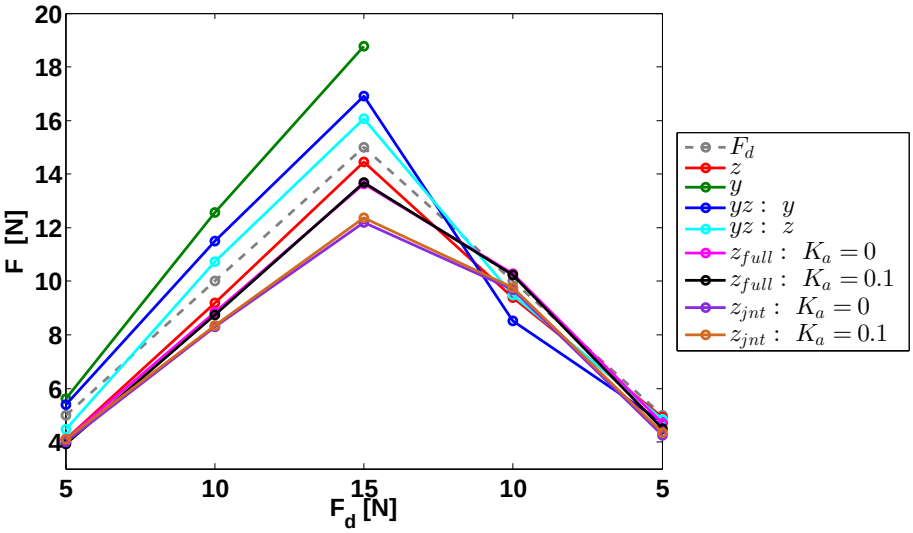


Figure 6.52: Overview of the averages of all experiments that apply a static force. Each dot represents the average of the measurement averages for a given desired force. This average considers all repetitions in all contact points, hence represent thirty measurement averages. The grey, dashed line represents the desired force. The labels indicate the direction of the desired force. The subscript states how this force is applied, other than expressing the constraints in Cartesian task space: *jnt* indicates expressing the constraints in joint space, *full* indicates using the full Cartesian task space to apply a wrench conforming to a force vector in the z-direction.

contact point, hence the possible arm movement when changing the desired force will be undesirable.

Figure 6.52 shows also the results of experiments without reference adaptation, i.e. $K_a = 0$. As expected, there is only a small difference between the results of the experiments with and without reference adaptation. Therefore, the experiments without reference adaptation are not analysed in detail here, but can be found in Appendix B.

Figure 6.53 gives an overview of the standard deviation of the averages of the experiments analysed in this section. The standard deviations of the measurements with a component in the y-direction have the highest standard deviations, mainly caused by the drift of some measurements. Using the full Cartesian task space to apply a wrench conforming to a force vector in the z-direction has a higher standard deviation than the other forces applied in

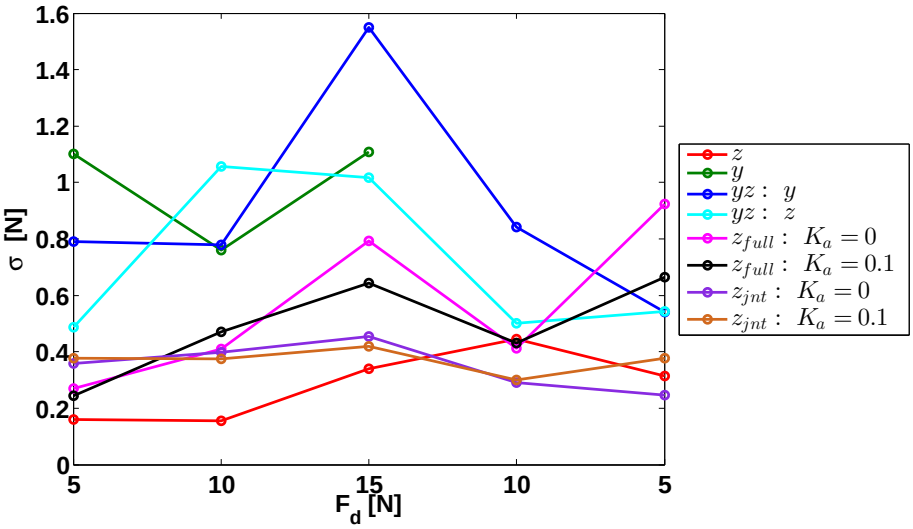


Figure 6.53: Overview of the standard deviation of the averages of all experiments that apply a static force. Each dot represents the standard deviation over all repetitions in all contact points, hence represent the standard deviation of thirty measurement averages.

the z-direction. A likely explanation is the unconstrained, hence non-constant contact point.

Further, the section analysed the use case of table wiping. The tool changes the dynamics of the robot arm, however this does not alter the control strategy. Since the robot arm moves while applying a force, the steady-state condition of the force control schemes does not hold for the robot joints. However, the difference between the applied and desired force is mainly dependent on the configuration of the robot joint, and less on the speed of the wiping.

This section shows that the control schemes introduced in Section 6.6 have a level of accuracy suitable for service robot tasks such as table wiping. A minimum desired force must be applied for the robot to overcome disturbances such as Coulomb friction. For the robot arm used in the here presented experiments a minimum value of around three Newton is required. The worst-case measured performance in the z-direction when expressing the force control task constraints in Cartesian task space has an error of $1.14N$ for a desired value of $5N$, or 23% of the desired value. However, this value can be decreased by first applying a higher force, and then reducing the force to $5N$ again. The worst-case measurement of the latter is $0.47N$ or about 10% of

the desired value. The worst-case measured performance in the y-direction is however higher, likely due to the non-gravity compensated gripper weight in the robot wrist joints. The worst-case performance in this direction is an error of $2.25N$ for a desired value of $5N$, or 45% of the desired value. The accuracy is robot configuration dependent, hence should be tested for the application at hand, and can possibly be compensated with an extra feedforward term.

6.8 Discussion

This section discusses the relation of the presented control schemes with the related work reviewed in Section 6.2.

6.8.1 Relation to resolved-velocity hybrid wrench/motion control

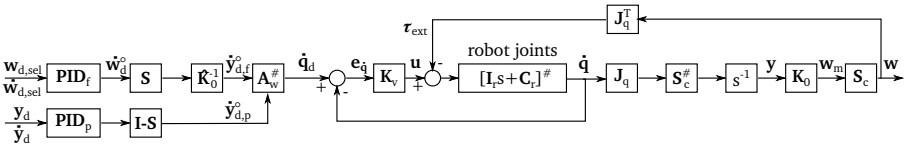


Figure 6.54: Resolved-velocity hybrid wrench/motion control related to the control schemes presented in this chapter. The PID blocks represent proportional controllers, and optionally include a feedforward, a derivative, or an integral term. These PID controllers use feedback signals from the system which are not shown on the figure. Control scheme adapted from [30].

Figure 6.54 shows an example resolved-velocity hybrid (wrench/motion) control scheme that relates to the control schemes presented in this chapter. Both the control schemes presented in this chapter, as well as the shown hybrid control scheme, consist of high bandwidth low-level velocity control loops, and high-level control loops that specify wrench and motion controlled directions in a generalized task space. The output of these high-level control loops define a desired velocity in the generalized task space $\dot{\boldsymbol{y}}_d^\circ$, which is transformed using the pseudo-inverse of the generalized Jacobian matrix $\boldsymbol{A}_W^\#$ to the desired joint velocities $\dot{\boldsymbol{q}}_d$. The latter serve as input to the low-level joint velocity controllers.

The force control aspect of resolved-velocity hybrid control typically defines a control law resulting in a desired derivative of wrench $\dot{\boldsymbol{w}}_d^\circ$. The latter is transformed to a desired velocity $\dot{\boldsymbol{y}}_{d,f}^\circ$ by multiplying by the inverse of the

estimate of the contact stiffness $\hat{\boldsymbol{K}}_0^{-1}$. $\dot{\boldsymbol{y}}_{d,f}^{\circ}$ defines the selected force/torque controlled directions of the total desired velocity $\dot{\boldsymbol{y}}_d^{\circ}$. As a result, resolved-velocity hybrid control needs an estimate of the contact stiffness. However, the approach introduced in this chapter does not require an estimate of the contact stiffness. It uses the inverse of the lower-level control gains, which have the dimensions of a mechanical damping, to transform a desired wrench to a desired velocity $\dot{\boldsymbol{y}}_{d,f}^{\circ}$.

6.8.2 Relation to resolved-velocity impedance control

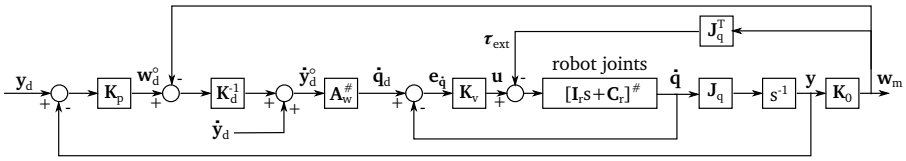


Figure 6.55: Resolved-velocity impedance control related to the control schemes presented in this chapter. Control scheme deduced from [168].

Figure 6.55 shows an example resolved-velocity impedance control scheme that relates to the control schemes presented in this chapter. Resolved-velocity impedance control is similar to the resolved-velocity hybrid (wrench/motion) control approach explained above. However, the wrench and motion components are not divided in separate controlled directions. Both wrench and motion control are expressed in a control loop resulting in a desired wrench $\boldsymbol{w}_d^{\circ}$. This desired wrench is transformed to a desired (generalized) task space velocity $\dot{\boldsymbol{y}}_d^{\circ}$ using the inverse of a damping matrix $\boldsymbol{K}_d$.

The control schemes presented in this chapter transform also the desired wrench to a desired (generalized) task space velocity $\dot{\boldsymbol{y}}_d^{\circ}$, using the feedforward factor $\boldsymbol{FF}$ which has the units of the inverse of mechanical damping. However, there are three major differences:

1. In the control schemes of this chapter, the desired wrench is a feedforward signal without integrating a desired motion,

2. In the control schemes of this chapter, the feedforward factor $\boldsymbol{FF}$ cannot be chosen freely, it is designed to cancel the low-level control dynamics. However, the control schemes presented in this chapter introduce a reference adaptation matrix $\boldsymbol{K}_a$, which in effect alters the low-level control gains and the feedforward factor $\boldsymbol{FF}$. This reference adaptation matrix $\boldsymbol{K}_a$

provides the possibility to adapt the dynamic behavior, but only within certain limits, as explained in Section 6.4.

3. In the control schemes of this chapter, there is no explicit feedback of the measured wrench. However, the contact wrench is implicitly estimated by the reference adaptation loop, using the joint velocity of the low-level velocity loop as a measure of the contact wrench.

6.9 Conclusions

This chapter introduced force control schemes that fit the resolved-velocity iTaSC control scheme, but which do not need a force sensor nor a precise dynamic model of the robot and its environment. The control schemes apply to robot-environment systems with finite stiffness, such that there is a relation between motion and force. They assume velocity-controlled robots which joint velocity errors reflect torque disturbances. In casu, a robot (i) that has backdrivable robot joints, (ii) for which the lower-level joint velocity control loops have known, only proportional gains, (iii) and for which these lower-level controllers run at sufficient high frequency such that their interactions can be neglected.

The control schemes, the abstract generalization of which is shown in Figure 6.2, feature a reference adaptation loop and a feedforward signal. The reference adaptation control loop feeds the joint velocity error $\boldsymbol{e}_{\dot{\boldsymbol{q}}}$ back to a task space controller. This joint velocity error presents a measure for the applied wrench, since this error is amplified by the gain $\boldsymbol{K}_p$ to the applied robot joint torques. In steady-state, these torques form an equilibrium with the force applied on the environment.[‡‡] In effect, the reference adaptation loop adapts the control gains of the robot low-level joint velocity controllers, in the selected directions of the task space. As a result, it alters the performance and damping of the controlled system in the selected directions. The feedforward signal takes the reference adaptation into account, hence the most important metric the reference adaptation factor K_a influences, is the dynamic response damping. Experimental validation shows the effectiveness of a well chosen K_a factor in imposing desired transient behavior on the applied force.

The chapter first analysed the special case of wrench nulling applied to human-robot comanipulation. Figure 6.6 shows the control scheme. In this use case, the robot has to follow the wrench applied by a human to the robot grippers. In this case, there is no feedforward signal and the reference adaptation loop

[‡‡] Neglecting unmodeled disturbances such as Coulomb friction.

reduces the damping felt by the human operator. Experimental validation shows that the robot provides following behavior by amplifying the applied wrench. The wrench nulling control scheme is integrated in a force-sensorless, bi-manual, human-robot comanipulation demo, which will be detailed in Chapter 7. A video [164][§§] shows how the robot follows the indications given by the human, using the presented control scheme.

Further, the chapter introduced, analysed, and experimentally validated two variations of a wrench control scheme, i.e. a scheme that tracks a desired wrench different from zero: one that expresses the force control task constraints in Cartesian task space, shown in Figure 6.25 and one that expresses these constraints in joint space, shown in Figure 6.27. The experimental validation shows that the control schemes are able to deliver a stable, constant contact force. The applied wrench is highly repeatable in a certain contact point, but its accuracy depends on the robot configuration. The experimental validation of the table wiping use case shows the applicability of the force control scheme to service robot tasks.

The experiments result in following guidelines: (i) Firstly, apply a small force setpoint to the system in order to avoid excitation of the robot-(sensor)-environment dynamics when making contact, (ii) secondly, increase the desired force to a level slightly higher than the actual desired value, (iii) thirdly, decrease the desired force to the actual desired value. The last two steps of the guidelines increase the accuracy of the applied force. The concrete trajectory that follows these guidelines does not have to be the five-step trajectory shown in the experiments: it can be replaced by another –possibly smooth– trajectory.

The introduced control schemes show some advantages over alternative approaches:

- The control schemes do not require a precise dynamic model of the robot, nor a model of the environment or the contact point, other than the minimal models needed to close the kinematic loops in iTaSC.[¶¶] As a consequence, the control scheme does not need contact detection. Activating the force control task results in an approach at constant velocity, contact, and a force build-up, without changing the control mode.
- The control schemes allow the combination of force control task constraints with other task constraints, using resolved-velocity constraint optimization.

[§§] Appendix C.3 lists and details the videos related to this chapter.

[¶¶] The models needed to close the kinematic loops in iTaSC can be minimal in the sense that they can contain uncertainty. For example when applying a force in a point on the table that is not enforced by any constraint. In this case, the frame defined on the table surface can have large uncertainties on all its DOF, without compromising the performance of the force controller.

- The control schemes avoid an expensive and complex force sensor, which is not always present on service robot platforms such as the PR2 robot.

Service robots need cost reduction, such as limiting the integrated sensors, to allow for mass commercialization. Further, service robots execute many concurrent tasks in unstructured environments. The presented control schemes fit these needs of service robot applications. Moreover, they prove to be of a level of accuracy that fits service robot tasks.

Future work should validate the approach on multiple robot platforms and analyse the platform dependency of observed effects. For example, it should analyse the effect of the increased accuracy in the applied force when decreasing the desired force setpoint. Furthermore, future work could optimize the performance on the PR2 platform by including numeric gravity compensation in the robot wrist joints, and adding an additional, configuration dependent force feedforward term. This feedforward term could compensate for the experimentally observed force offsets.

Chapter 7

Force-Sensorless and Bimanual Human-Robot Comanipulation*

7.1 Introduction

This chapter demonstrates the lessons learned from the previous chapters of this dissertation, applying the code support discussed in Chapter 4 and 5, and the wrench nulling control scheme of Chapter 6, to a force-sensorless† and bimanual human-robot comanipulation task. In this application a PR2 robot: (i) co-manipulates an object with a person with its two grippers, (ii) avoids dynamic and static obstacles with its base, (iii) maintains visual contact with the operator, and (iv) prevents unnatural poses. This application, shown in Figure 7.1, comprises three key challenges: (*i*) the definition of several constraints in different control spaces for a high DOF robot setup (20 DOF in total), (*ii*) the implementation of this task in a reusable software framework, and (*iii*) the implementation of a direct human-robot interaction task without the use of a force sensor. Preliminary results of the application were shown at the "Standard Platform Demonstration" booth at IROS 2011.

*This chapter is partially based on Vanthienen, D., De Laet, T., Decré, W., Bruyninckx, H., De Schutter, J. (2012), "Force-Sensorless and Bimanual Human-Robot Comanipulation", *10th IFAC Symposium on Robot Control*, Dubrovnik, Croatia, 5-7 September 2012 (pp. 1-8)

†This is preferred over estimating the forces by measuring the currents in the actuators, because these estimates are disturbed by joint friction forces

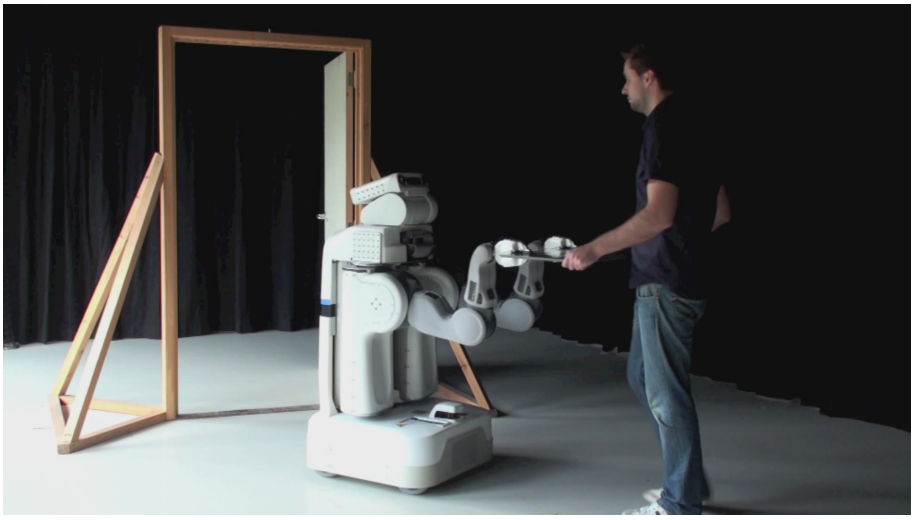


Figure 7.1: Force-sensorless human-robot comanipulation. A robot helps a human carrying a plate from one side of the door to the other, while avoiding to hit the door, maintaining visual contact with the operator, and avoiding unnatural poses.

Different approaches to estimate the external wrench based on a disturbance observer are presented in literature [59, 84, 151]. These approaches require a precise dynamic model of the robot. Since such a model is unavailable for the PR2, and the application does not require accurate force control, we use a simple estimation scheme.

The application was first developed using an earlier version of the iTaSC software framework [165, 166]. This development followed the conceptual iTaSC workflow, discussed in Section 2.2.2. The application proved to be hard to develop, and the result difficult to maintain or adapt. The experiences of creating this use case triggered the developments and insights of this dissertation. These new insights where then re-applied to the use case.

The chapter is organized as follows: Section 7.2 applies the modelling procedure explained in Section 2.2.2 to the bimanual human-robot comanipulation application. Section 7.3 presents the experimental results showing the successful application of iTaSC on the comanipulation application. Section 7.4 discusses how the application of the insights and software tools of this dissertation to the application improved its quality. Finally, Section 7.5 concludes the chapter by summarizing the contributions and discussing future work.

7.2 iTaSC modelling of the comanipulation application

This section details the iTaSC modelling of the bimanual human-robot comanipulation application following the workflow explained in Section 2.2.2. In particular, it discusses the different components of the iTaSC application: the robot and objects (Section 7.2.1), the tasks (Section 7.2.2), the world model (Section 7.2.3), and the solver (Section 7.2.4).

7.2.1 Robots and objects

The application involves one PR2 robot and two objects: an obstacle and a moving person.

The **PR2 robot** has a tree-structured model, consisting of a branch from a reference frame $\{b\}$ fixed to the world through the robot base to its spine, from where the two seven DOF arm branches and the two DOF head branch start. Object frames $\{o\}$ are defined on the PR2's grippers, base, and head. As the PR2 robot moves in the application scene, its pose (i.e. the entire kinematic chain of the PR2) in the world model is continuously updated using the PR2's odometry and encoders.

The **obstacle**, which the PR2 has to avoid in the application at hand, is considered fixed in the scene for sake of simplicity, but could be moving and/or uncertain. Therefore the obstacle's reference frame $\{b\}$ is fixed to the world. In this case a model with uncertainty coordinates $\boldsymbol{\chi_u}$ can be included with or without sensor measurements and motion model to update the obstacle's pose. An object frame $\{o\}$ is attached at the centre of the obstacle, with a fixed transformation with respect to the obstacle's reference frame $\{b\}$.

The position of the **moving person** is estimated by a face detection algorithm, applied on the robot's camera images. The face detection algorithm estimates the position of the head, which is then used to update the position of the moving person, and of the moving person's reference frame, in the world model. An object frame $\{o\}$ is defined on the head of the person with a fixed transformation with respect to the moving person's reference frame.

7.2.2 Tasks

As explained in Section 2.2.2, a task consists of the definition of a VKC between object frames (*feature space*), *constraints* on the outputs $\boldsymbol{y}$ of the task's kinematic

loop, *controllers* enforcing the different constraints, and *set-point generators* delivering the desired output values to the controllers. This section specifies and details all the tasks involved in the application.

The bimanual human-robot comanipulation application involves different tasks. Since the PR2 has to manipulate a rigid object with its two hands, a **grippers-parallel task** keeps the grippers parallel such that both hands are not moving with respect to each other and with respect to the manipulated object. Furthermore, the PR2 should comanipulate the object with a human, and therefore should react on the wrenches the human is applying to the object. Implementing this reaction to wrenches applied by the human results in a following behavior. To this end **a wrench-nulling task for each hand** is specified, i.e. a task whose goal is to minimize the wrench applied by the human. During the comanipulation the PR2's base has to avoid a fixed obstacle in the scene, specified by an **obstacle-avoidance task** and the PR2 head should watch the human operator, specified by a **head-tracking task**. Finally, a **joint-limits task** is defined to keep the robot joints away from the limits. The following paragraphs elaborate on the task definitions.

Grippers-parallel task

The grippers-parallel task defines a *Cartesian feature space* between the object frame on the left gripper $\{o1^p\}$ and the object frame on the right gripper $\{o2^p\}$. The first feature frame $\{f1^p\}$ coincides with $\{o1^p\}$ and the second feature frame $\{f2^p\}$ coincides with $\{o2^p\}$. The feature coordinates defining the six DOF of the resulting *VKC* are all located between $\{f1^p\}$ and $\{f2^p\}$, i.e. $\boldsymbol{\chi_{fI}}^p = 0$ and $\boldsymbol{\chi_{fIII}}^p = 0$, while an intuitive definition of $\boldsymbol{\chi_{fII}}^p$ is obtained by expressing these coordinates in the first feature frame $\{f1^p\}$:

$$\boldsymbol{\chi_{fII}}^p = [x^p, y^p, z^p, \phi^p, \theta^p, \psi^p]^T , \tag{7.1}$$

where $[x^p, y^p, z^p]^T$ define the $3D$ position coordinates of the second gripper with respect to the first gripper and $[\phi^p, \theta^p, \psi^p]$ is a set of Euler-angles defining the orientation between the second and first gripper.

The task *constrains* all DOF of the VKC in order to keep the full pose between the grippers fixed. Therefore, the output vector $\boldsymbol{y}^p$ equals the feature coordinate vector $\boldsymbol{\chi_f}^p$. A *proportional controller* is used to achieve the desired fixed pose, which is delivered by a *set-point generator* simply generating a fixed value ($x_d^p = 0m$, $y_d^p = 0m$, $z_d^p = 0.3m$, $\phi_d^p = 0rad$, $\theta_d^p = 0rad$, $\psi_d^p = 0rad$). The full rotation matrix is used in the control law to prevent singularities.
Figure 7.2 shows the kinematic loop of this task.

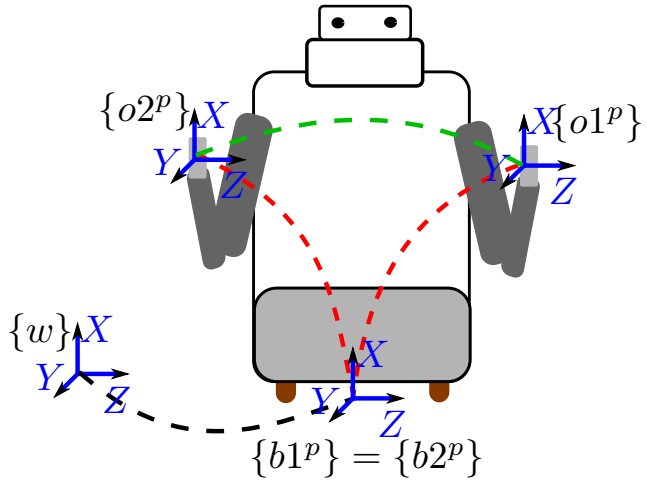


Figure 7.2: Kinematic loop of the **grippers-parallel task** (dashed lines). The red lines represent the kinematic chains of the robot and objects, the black line represents the (fixed) relation between the robot's fixed reference frame ($\{b\}$) and the world reference frame ($\{w\}$) defined in the world model, and the green lines represent the *VKC* between the object frames ($\{o1\}$ and $\{o2\}$).

Wrench-nulling tasks

The applications contains a wrench-nulling task for each gripper, nulling the wrench exerted on that gripper. One wrench-nulling task defines a *Cartesian feature space* between the object frame on a gripper $\{o1^n\}$ and an object frame coinciding with the robot its mobile base $\{o2^n\}$. The first feature frame $\{f1^n\}$ coincides with $\{o1^n\}$ and the second feature frame $\{f2^n\}$ coincides with respect to $\{o2^n\}$. The feature coordinates defining the six DOF of the resulting *VKC* are all located between $\{f1^n\}$ and $\{f2^n\}$, i.e. $\boldsymbol{\chi_{fI}}^n = 0$ and $\boldsymbol{\chi_{fIII}}^n = 0$, while an intuitive definition of $\boldsymbol{\chi_{fII}}^n$ is obtained by expressing these coordinates in the second feature frame $\{f2^n\}$:

$$\boldsymbol{\chi_{fII}}^n = [x^n, y^n, z^n, \phi^n, \theta^n, \psi^n]^T \,, \tag{7.2}$$

where $[x^n, y^n, z^n]^T$ define the $3D$ position coordinates of the gripper with respect to the robot base and $[\phi^n, \theta^n, \psi^n]$ is a set of Euler-angles defining the orientation between the gripper and the robot base.

The output we are interested in is the wrench $\boldsymbol{y}^n = \left[F_x^n, F_y^n, F_z^n, M_x^n, M_y^n, M_z^n\right]^T$, which is considered in a *Cartesian feature space* between the object frames on respectively the robot gripper and the robot base. Section 6.3 explains how the

wrench output can be controlled using the wrench-nulling control scheme shown in Figure 6.8 and the available task coordinates $\boldsymbol{\chi_f}^n$. Section 6.3.2 details the experimental validation of this control scheme as part of the here presented comanipulation application. Figure 7.3 shows the kinematic loop of this task.

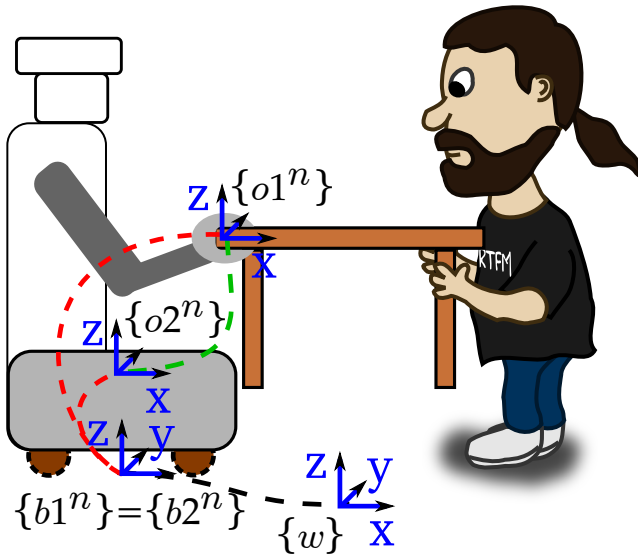


Figure 7.3: Kinematic loop of a **wrench-nulling task** (dashed lines). The red lines represent the kinematic chains of the robot and objects, the black line represents the (fixed) relation between the robot's fixed reference frame ($\{b\}$) and the world reference frame ($\{w\}$) defined in the world model, and the green lines represent the *VKC* between the object frames ($\{o1\}$ and $\{o2\}$).

Obstacle-avoidance task

The obstacle-avoidance task for the robot base, defines a feature space with a cylindrical coordinate system, between the object frame on the obstacle $\{o1^o\}$ and the robot's base frame $\{o2^o\}$. The first feature frame $\{f1^o\}$ coincides with $\{o1^o\}$ and the second feature frame $\{f2^o\}$ coincides with respect to $\{o2^o\}$. The feature coordinates defining the six DOF of the resulting *VKC* are all located between $\{f1^o\}$ and $\{f2^o\}$, i.e. $\boldsymbol{\chi_{fI}}^o = 0$ and $\boldsymbol{\chi_{fIII}}^o = 0$, while an intuitive definition of $\boldsymbol{\chi_{fII}}^o$ is obtained by expressing these coordinates in the first feature frame $\{f1^o\}$ using a cylindrical coordinate system:

$$\boldsymbol{\chi_{fII}}^o = [\theta^o, r^o, z^o, \alpha^o, \beta^o, \gamma^o]^T, \tag{7.3}$$

where $[\theta^o, y^o, z^o]^T$ define the $3D$ position coordinates of the robot base with respect to the obstacle and $[\phi^o, \theta^o, \psi^o]$ is a set of Euler-angles defining the orientation between the robot base and the obstacle.

As a result the distance to the object is represented by a single coordinate r^o. Since the distance is the only value of interest, only this DOF is *constrained*. Therefore, the output y^o is equal to r^o.

An inequality constraint is defined for the output, keeping the robot at a safe distance of the obstacle: $y^o \geqslant D$. To implement the inequality constraint, we use an approach using equality constraints enforced by a proportional controller and constraint monitoring. This approach uses adaptive constraint weights, which can gradually increase (decrease) to activate (deactivate) the constraints depending on (near) constraint violation. When the robot approaches the obstacle closer than a configured value, the distance constraint is activated. Consequently a *set-point generator* delivers a, in this case fixed, desired distance to the object (a little bit bigger than the D in order to drive the robot away from the obstacle.). When the robot is far enough from the obstacle, the task is deactivated by putting the task weight to zero. The `Coordinator` of this task handles the coordination of the constraint weights. Figure 7.4 shows the kinematic loop of this task.

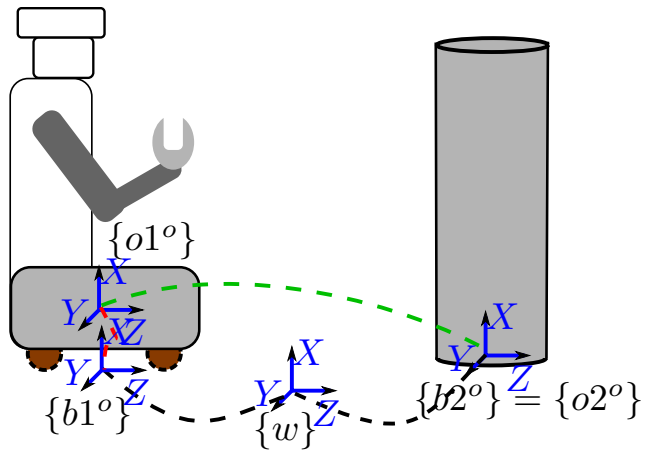


Figure 7.4: Kinematic loop of the **obstacle-avoidance task** (dashed lines). The red lines represent the kinematic chains of the robot and objects, the black line represents the (fixed) relation between the robot's fixed reference frame ($\{b\}$) and the world reference frame ($\{w\}$) defined in the world model, and the green lines represent the *VKC* between the object frames ($\{o1\}$ and $\{o2\}$).

Head-tracking task

The head-tracking task defines a *Cartesian feature space* between the object frame on the robot's head $\{o1^h\}$ and the object frame on the head of the person $\{o2^h\}$.

The first feature frame $\{f1^h\}$ coincides with $\{o1^h\}$ and the second feature frame $\{f2^h\}$ coincides with respect to $\{o2^h\}$. The feature coordinates defining the six DOF of the resulting *VKC* are all located between $\{f1^h\}$ and $\{f2^h\}$, i.e. $\boldsymbol{\chi_{fI}}^h = 0$ and $\boldsymbol{\chi_{fIII}}^h = 0$, while an intuitive definition of $\boldsymbol{\chi_{fII}}^h$ is obtained by expressing these coordinates in the first feature frame $\{f1^h\}$:

$$\boldsymbol{\chi_{fII}}^h = \left[x^h, y^h, z^h, \phi^h, \theta^h, \psi^h\right]^T, \tag{7.4}$$

where $[x^h, y^h, z^h]^T$ define the $3D$ position coordinates and the Euler-angles $[\phi^p, \theta^p, \psi^p]$ define the orientation, between the person's and the robot's head.

The z^h-direction connects the two object frame origins, i.e. the PR2's and the person's head. The task *constrains* the x^h and y^h coordinates, perpendicular to the z^h direction, to be zero. Therefore, the output vector $\boldsymbol{y}^h$ is equal to $\left[x^h, y^h\right]^T$. A *set-point generator* delivers a zero value for both the x^h and y^h coordinate, enforced by a *proportional controller*. As a result, the head of the robot will align with the head of the person, hereby keeping the PR2's cameras pointed towards the person's head. Figure 7.5 shows the kinematic loop of this task.

Joint-limits task

The joint-limits task keeps the PR2's arm joints out of its limits. It *constrains* the robot joints themselves, therefore there is no need for extra loop closure equations and hence *no VKC*. As for the obstacle avoidance task, an equality constraint approach is chosen. The `Coordinator` of the task activates the joint-limits task when a joint approaches its lower joint limit q_{min} or higher joint limit q_{max} by a configured distance, i.e. when it reaches $q_{min,m}$ or $q_{max,m}$, The task applies a velocity to the joint to move it away from the approached limit. When the joint position is far enough from its limits, the `Coordinator` deactivates the task. The magnitude of the desired velocity $\dot{q}_d$, delivered by the *set-point generator*, as well as the weight W of the task, increase with the proximity to the joint limits, as shown in figures 7.6 and 7.7. As a result, the activation of the joint-limits task causes a smooth transition, rather than a discontinuity in behaviour.

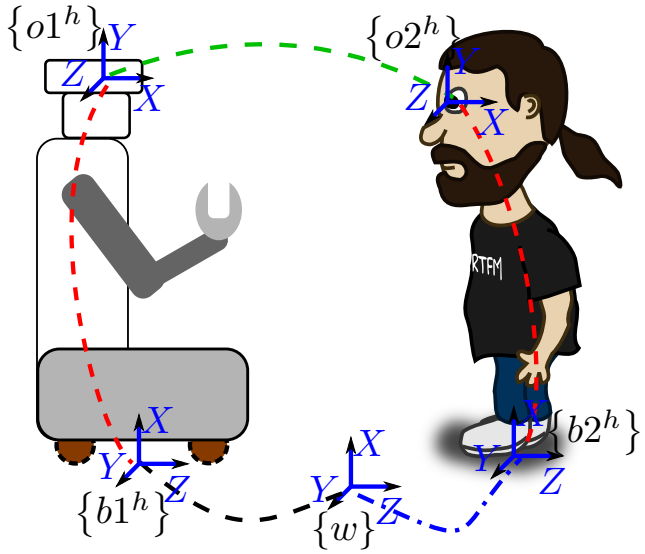


Figure 7.5: Kinematic loop of the **head-tracking task** (dashed lines). The red lines represent the kinematic chains of the robot and objects, the black line represents the (fixed) relation between the robot's fixed reference frame ($\{b\}$) and the world reference frame ($\{w\}$) defined in the world model, and the green lines represent the *VKC* between the object frames ($\{o1\}$ and $\{o2\}$).

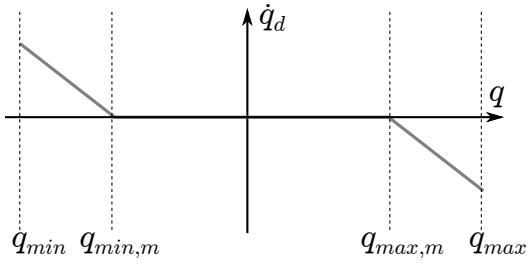


Figure 7.6: Desired joint velocity $\dot{q}_d$ in function of the joint position q. q_{min} and q_{max} define the upper and lower joint limits respectively.

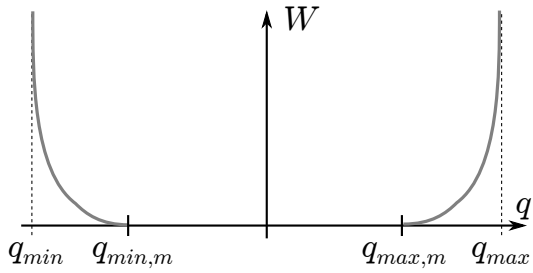


Figure 7.7: Desired joint weight of the joint-limits task in function of the joint position q. q_{min} and q_{max} define the upper and lower joint limits respectively.

7.2.3 The world model

The world model keeps track of all robots and objects involved in the application, and in particular of their poses with respect to the world reference frame $\{w\}$. The scene component of the software support contains this world model. Other components (the robots and objects, the tasks and the solver) of iTaSC deliver their status information, e.g. current pose, desired velocities, ..., to this scene component and request similar world model information from it.

7.2.4 The solver

The solver calculates the desired joint velocities $\dot{\boldsymbol{q}}_d$, to be sent to the controllable DOF of the robots by solving an optimization problem involving the task constraints, the constraint weights, the task priorities, and the robot weight matrix. In this application the solver has to calculate the desired velocities for the 20 controllable DOF of the PR2, out of maximum 35 constraint equations. By taking weights and priorities of the different task constraints and the robot DOF into account, the behaviour of the application can be configured.

The application uses the prioritized, weighted, damped pseudo-inverse algorithm discussed in Section 2.2.3 as solver. This pseudo-inverse is denoted $\boldsymbol{A}_W^{\#}$, where $\boldsymbol{A}$ is the augmented Jacobian relating the combined task spaces and the robot joint space. Figure 7.8 shows the overall control scheme of all tasks combined and the solver calculating the joint velocities.

In this application, all tasks have the same priority. The robot joint weights are well-chosen to achieve natural human-robot comanipulation behaviour: putting higher weights on the PR2's base DOF than on the PR2's arm and head DOF, will favour arm and head motions over base motions. As a consequence, when

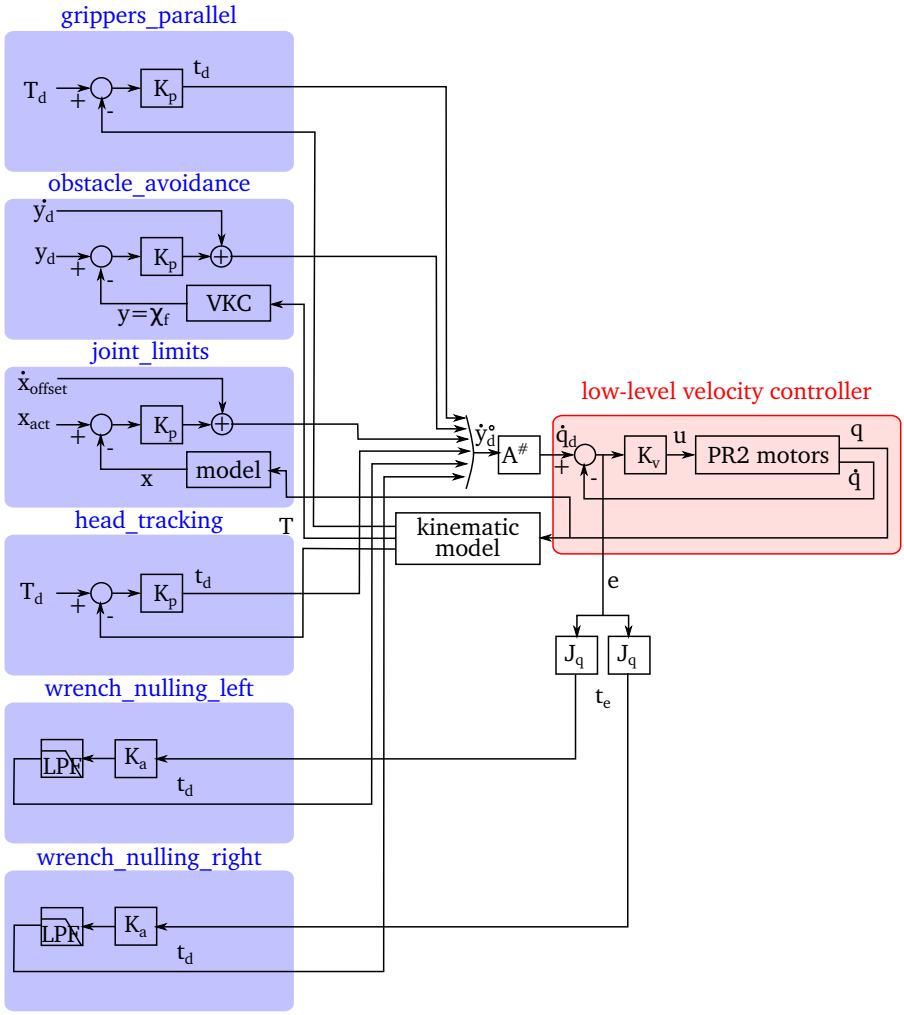


Figure 7.8: Control scheme of the full application. Purple boxes indicate the controllers in the different generalized task spaces, and the red box indicates the low-level robot joint velocity controller. $\boldsymbol{K}_a$ indicates the reference adaption factor, $\boldsymbol{K}_p$ indicates different proportional gains, $\boldsymbol{A}_W^{\#}$ indicates the weighted, damped pseudo-inverse of the augmented Jacobian, and $\boldsymbol{J}_q$ indicates the Jacobian relating the wrench-nulling task space and the robot joint space.

pulling an arm of the robot, the robot will tend to move its arms first and only when approaching joint limits, its base. Section 7.3 elaborates on the interaction of the different tasks.

The resulting desired joint velocities $\dot{\boldsymbol{q}}_d$ are sent to the PR2's standard low-level velocity controller of the robot_mechanism_controllers package, as available on ros.org [171].

7.3 Results

The bimanual human-robot comanipulation application consists of the joint execution of all described tasks: wrench-nulling, keep grippers parallel, keep visual contact with the operator, avoid joint limits, and avoid obstacles. A video [164] proves the performance of the full task with all described constraints active in different scenarios.

This section provides quantitative results of some key interactions. To this end, we have set up a reduced, but repeatable experiment involving a subset of tasks: avoid joint limits, wrench-nulling, and obstacle avoidance. While the second task is always active, the first and the third task show the ability to activate and deactivate, as well as to change the weights of different constraints in a stable way.

Section 6.3.2 discusses the experimental validation of the wrench-nulling task. It shows that applying a constant force to a gripper of the robot results in a constant assistance, after transition behavior.

7.3.1 Experimental setup

A cable connects a mass of 1.5kg to the robot's left hand through a pulley system. When releasing the mass, the weight of the mass pulls the hand with a constant force in a direction approximately aligned with the x-direction. The robot arm joints are far from their limits before applying the force, with the elbow pointing downwards. Figure 7.9 depicts the experimental setup.

7.3.2 Joint limit activation

First consider a wrench-nulling task and joint-limits task on the left arm, both with the same priority. When applying a constant force to the robot's left hand, the hand will move in the direction of the applied force. This movement uses

the available DOF in a way that depends on the weight and priority of each task. The weight (*penalty*) on the DOF of the robot's mobile base is set higher than the DOF of the arms.

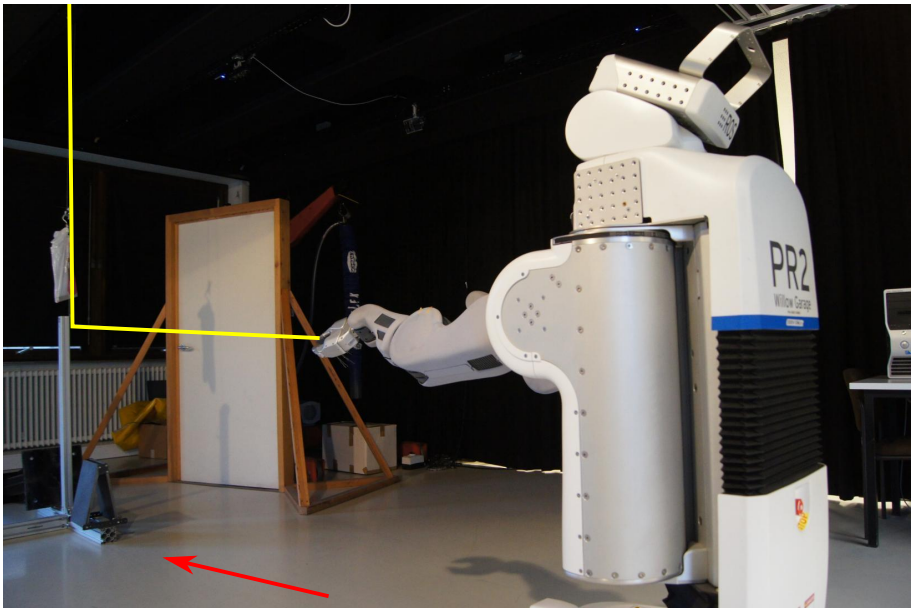


Figure 7.9: Picture of the experimental setup. The cable (yellow) is enhanced for visibility. The cable and pulley system connects the PR2 robot's hand to the white bag with the weight, shown at the left. The the red arrow indicates the x-direction.

Figure 7.10 shows the position of the left elbow joint and the x-coordinate of the base over time, after releasing the mass. Both the elbow and the base move, at a ratio determined by the weights on the DOF of the robot. The base does not start moving at the same instant as the arm due to a threshold implemented to prevent nervous base behavior. The elbow joint reaches its joint limit of -0.82rad after about 2s, as indicated by the red dash-dotted line in Figure 7.10. This event activates the joint-limit task that pushes the joint position away from its limit. The elbow joint finds an equilibrium between the joint-limit task and the wrench-nulling task at -0.78rad. From that moment on, the robot's base satisfies the wrench-nulling constraint by moving faster. Figure 7.10 shows this faster increase of the base position after the red dash-dotted line.

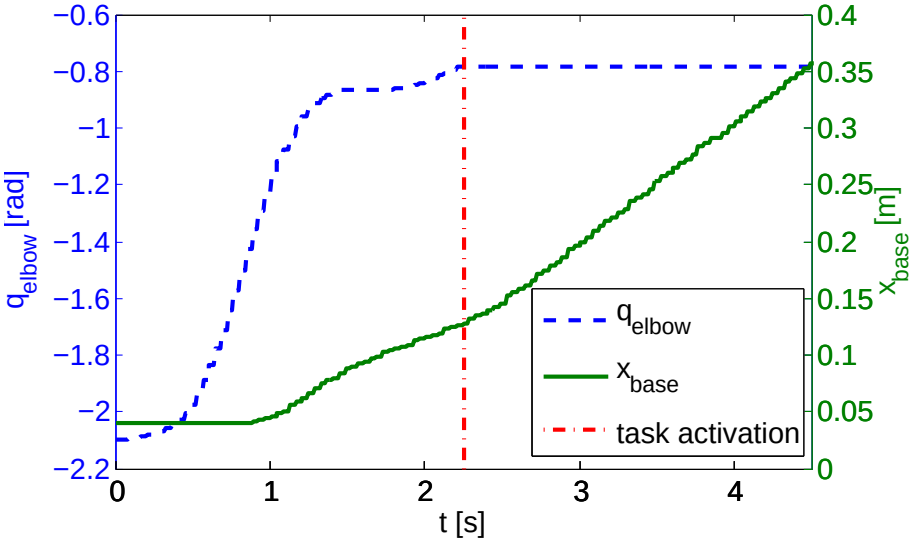


Figure 7.10: Elbow and base position over time, in blue dashed and full green line respectively. The red dash-dotted line indicates the moment the joint limit constraint is activated on the left elbow joint.

7.3.3 Obstacle avoidance

An obstacle is added to the setup of the wrench-nulling experiment at coordinates $(x, y) = (1.3m, -0.1m)$ with respect to $\{w\}$, as shown in Figure 7.11. Adding an obstacle avoidance task, prevents that the robot base collides with the obstacle. The task constrains the robot base to keep a distance r^o of 0.6m when approaching the object closer than 0.5m.

Figure 7.12 shows the path of the robot in the x, y-plane with a blue dashed line. The red full lines indicate the parts of the path where the obstacle avoidance task was active during the experiment. The green dashed circle segment indicates the obstacle, the magenta dash-dotted circle segment indicates a distance r of 0.5m from the obstacle center. As can be seen, the robot moves first along a line in the direction of the applied force. When the robot's base approaches the obstacle, the obstacle-avoidance task is activated and hence the the robot's base avoids a collision, while still trying to follow the wrench and joint-limit constraints.

However, the performance of the application can still be improved. The current implementation solves the optimization problem instantaneously, without taking future desired task values into account. This causes nervous behavior of the

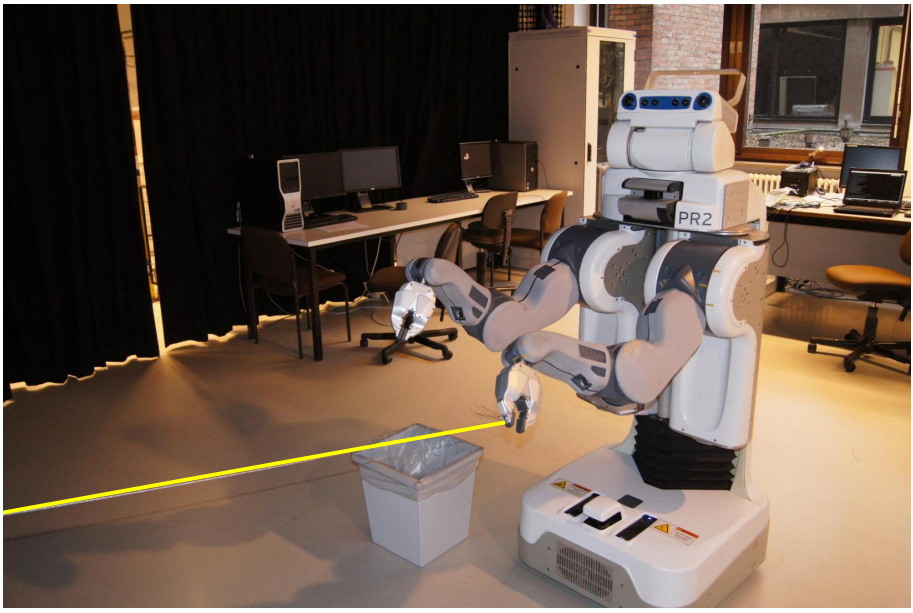


Figure 7.11: Picture of the PR2 avoiding the recycle bin. The cable (yellow) is enhanced for visibility.

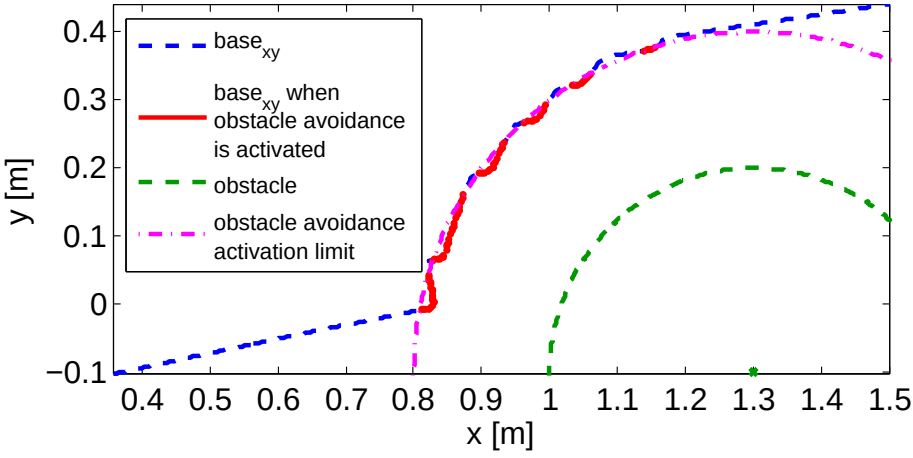


Figure 7.12: The blue dashed line indicates the path of the robot base in x, y-plane. The red full lines indicate the parts of the path where the obstacle avoidance task was active. The green dashed circle segment indicates the position of the obstacle, the magenta dash-dotted circle segment the distance r at which the obstacle avoidance task is activated.

mobile base of the robot, which consists of four actuated wheels that can be rotated. This rotation takes time, which can not be taken into account in the current instantaneous solver. Changing the `Solver` to an optimization-problem solver that can take into account the desired values over a time horizon could improve performance. Decré et al. [51] described such a solver, which is however not supported by the current implementation of the iTaSC software framework.

Another point of improvement are better joint velocity measurements. These measurements, delivered by the platform, have currently a rather low signal-to-noise ratio. As a result, a low-pass filter is currently needed on these signals.

7.4 Recreating the use case using the Composition Pattern and the DSLs

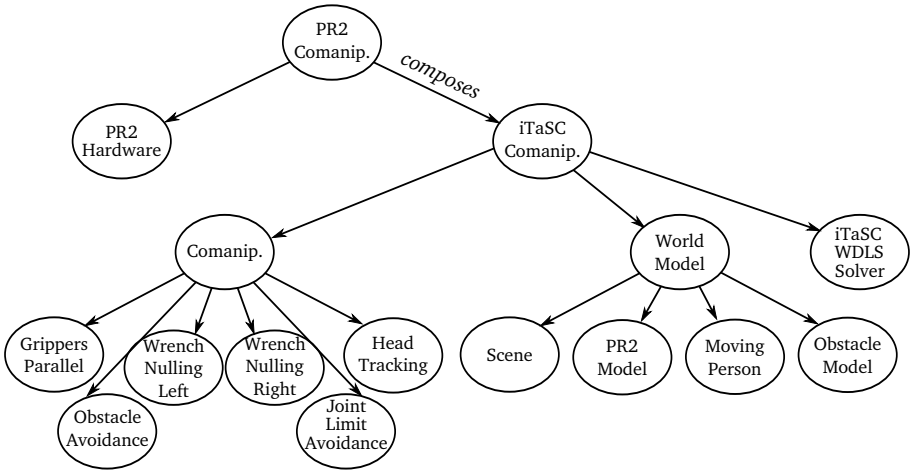


Figure 7.13: Composition tree for the force-sensorless human-robot comanipulation application with a PR2 robot. Each node represents a model of a (Composite) `Functional Entity`, which conforms to the meta-model expressed in the corresponding node in Figure 3.5 of Chapter 3. Each of the `Task` entities is a Composite `Functional Entity` of which the composition is not shown in the figure. The word 'comanipulation' is abbreviated to 'comanip.'

The first implementation of the presented use case followed the iTaSC workflow, as detailed in Section 2.2.2, and used the supporting components of an earlier version of the iTaSC software framework, which is mainly built on top of Orocos [33] and rFSM [88].

Later, a person not involved in the development of the constraint-based programming DSL of Chapter 4 was asked to recreate this implementation using this DSL.

The following sections (i) describe the recreation of the application, and (ii) discuss the difference in development and code of the recreated application with the original application.

7.4.1 Recreation of the application using the constraint-based programming DSL

A person not involved in the development of the constraint-based programming DSL of Chapter 4 was asked to recreate the comanipulation application described in this chapter in order to test

- whether the DSL is easy to understand and use by someone with a background in constraint-based programming, and
- whether such a person is able to create constraint-based programming applications faster.

More concretely, a summer intern was asked to (i) make himself familiar with the constraint-based programming DSL introduced in Chapter 4 (and the DSLs it integrates), and (ii) subsequently recreate the comanipulation application explained above. The intern was a master student in mechanical engineering at our department. Prior to his internship, he did experiments on the force-sensorless wrench control scheme of Chapter 6, and he followed a course on constraint-based programming and the iTaSC workflow. He was given the design of the application as explained in Section 7.2 and the original code. The latter was, however, outdated and not working at that time.

The intern used the DSL as a template to model the application. Figure 7.13 shows the composition tree (structure) of the resulting model. Each of the non-leaf entities in this tree has a `Coordinator`, `Configurator` and `Composer`, as explained in Chapter 4. Each of the `Task` models shown in Figure 7.13 is a Composite `Functional Entity` composing following `Functional Entities`:

- a `Constraint-Controller`, which models a proportional controller for all tasks except the *wrench-nulling* `Tasks`, which use an implementation of the wrench-nulling control scheme of Figure 6.8;

- a `Virtual Kinematic Chain`, which models the Virtual Kinematic Chains as described in Section 7.2, except for the *joint-limits* `Task`, which only imposes constraints on the robot joints;
- one or two `Setpoint Generators`, which model the joint velocity and weight trajectories to follow (for the *obstacle-avoidance* and *joint-limits* `Task`) or the setpoints to maintain (for the *grippers-parallel* and *head-tracking* `Task`). Figures 7.6 and 7.7 show the trajectories for the *joint-limits* `Task`. The *wrench-nulling* `Tasks` do not have a `Setpoint Generator`.[‡]

Since the comanipulation application uses the same `Platform` as the drawer opening application that was explained in Chapter 4, the intern could reuse the model as stated in Listing 4.1 (Chapter 4) with only minor changes. Also the `Constraint-Based Program` is very similar to the `Constraint-Based Program` of the drawer opening application, shown in Listing 4.2 (Chapter 4). Only the composite `Task` and `World Model` needed to be replaced and their `Composer` changed to connect the `Task` and `World Model`.

Next, the intern had to create (i) a new `World Model` that composes the *Scene*, *PR2 Model*, *Moving Person*, and *Obstacle Model* `Functional Entities`, (ii) a `Coordinator` for which a template from the iTaSC software framework could be used, (iii) a `Configurator` to apply the model configurations, and (iv) a `Composer` to attach the `Robots` and `Objects` to the `Scene`.

Subsequently, he had to create the *comanipulation* composite `Task`, which composes (i) the different `Task Functional Entities` shown in Figure 7.13, (ii) a `Coordinator`, (iii) a `Configurator`, and (iv) a `Composer`. The intern could use a template from the iTaSC software framework for the `Coordinator`. In this template, only the events to activate all tasks in the 'running' state (Section 5.7.1) needed to be filled in, since all tasks are always active once the `Application` is running. Furthermore, the intern had to develop a `Configurator` which sets the `Task` weights and priorities, and a `Composer` for which a template from the iTaSC software framework could be used.

Each `Task` composes the `Functional Entities` described above, next to a `Coordinator`, a `Composer`, and a `Configurator`. `Monitors` in the form of Orocos plug-ins where included to (i) monitor the robot joints for the *joint-limits* `Task` and (ii) the distance between the robot base and the obstacle, i.e. one of the feature coordinates of the *obstacle-avoidance* `Task`. The `Coordinators` and (large part of) the `Composers` of the different tasks could

[‡]At the time, the *wrench-nulling* `Tasks` did have a '`Setpoint Generator`' in order to receive the robot joint velocity errors needed in the controller, since the iTaSC software framework did not provide the functionality to receive this data.

be reused from libraries of existing `Tasks` in the iTaSC software framework. These include `Coordinators` that activate and deactivate the *joint-limits* and *obstacle-avoidance* `Tasks` based on the events from the `Monitors`. However, the configuration in the different `Configurators` needed to be defined for each `Task`. Configuration of the original application code formed a start point, but needed to be adapted. This step, which includes fine tuning the `Constraint-Controllers` of the different `Tasks`, took the majority of the development time. Although all `Tasks` are presented here in one scheme, the intern had to try each `Task` separately before composing it with other `Tasks`, to form ultimately the whole of the `Application`.

It is a desired outcome that majority of the development time is spent on determining configuration (parameter) values and testing of the application, since these steps are not automated or made easier by the DSL.

Section 4.7 discusses the lines of code involved in the here described application modeling.

7.4.2 Discussion

The first implementation of the presented application used the general tooling from Orocos to instantiate the iTaSC application. This tooling is not specialized for the application or domain of constraint-based programming. This resulted in a *single large file*[§] to instantiate ('deploy') an application, which we will further refer to as the deployment file. Moreover, the configuration of the application was (i) or one large file, (ii) or spread out to different, procedural dependent locations, i.e. dependent on the phase in the overall life-cycle where the configuration was needed. For example the configuration was located in the deployment file, a configuration file of a component or a finite-state machine. The Composition Pattern gave each concern, including the configuration, a single, **clearly defined place in the overall structure**.

Furthermore, consistency depended on names, used throughout the different files, effectively introducing *multiple points of name introduction*. For example, to change a task, one had to

- load the library containing the components relating to the task,
- create the necessary components,
- configure the components,

[§]We will refer to a piece of code as a 'file', highlighting the location of the code. This piece of code can be written in different languages.

- connect the component to a timer
- connect the data ports of the component to its peers[¶], etc.

All these steps refer to the components by name, which have to match for each instruction. All of these steps are still necessary in the current framework, however the model of an application, written in the DSL presented in Chapter 4, forms a single, structured source of information, without multiple points of name introduction. As a result, the different parts of the current implementation are **easier to keep consistent, and are more flexible.**

Furthermore, the earlier implementation of the application was *hard to debug*: most errors, e.g. a wrong name of a component reference, occur only at construction or run-time. The DSL enabled **model verification**, returning **useful errors before deployment**.

Moreover, an iTaSC application was created in a procedural way, partially at run-time: a succession of instructions (e.g. method calls) to create the application, each of which has to check consistency of information before proceeding to the next. This showed a clear *problem of composition*. For example, structuring and configuring a task and its connections, i.e. between a virtual kinematic chain, controller, and setpoint generator, was handled by a sub-state machine of the application. Moreover, it showed a large interconnection of *application specific and framework specific* functionality. The introduction of the **metamodeling approach resolved this interconnection**.

The first implementation of the use case did separate coordination from the functionality, however the coordination used a FSM that incorporated framework specific code, and did not separate the five concerns properly. For example, when the PR2 robot approaches an obstacle, the FSM transitions to a state that invokes and then sets the Orocos Property that activates the obstacle avoidance task. Similarly, the FSM can write to an Orocos port or call an Orocos operation. This proved a very *rigid structure*, only applicable for this framework and concrete set of components. **This rigidity was resolved by the separation of the 5Cs using the Composition Pattern**.

Furthermore, to create the use case using the iTaSC software framework **required knowledge** of the framework, but also different underlying frameworks and software languages, such as C++, Orocos (the framework, but also the integrated scripting languages), ROS, etc. In contrast, the use case was recreated by a summer intern with limited knowledge of these languages and frameworks. Although the iTaSC software framework can still be improved in many ways, e.g. separation of the functionality from the framework, the

[¶]The framework did provide minimal automation for connections.

DSL made these improvements an 'implementation detail' from the application developer's perspective.

The iTaSC workflow still forms a good general guideline to create iTaSC applications, however focusses only on the functionality (computation). The presented DSL is more detailed, giving a **template for all concerns**. Furthermore, due to the separation of concerns and the domain modeling, it is easier to design each of the separate entities in a more flexible way. Moreover, the DSL brings the design workflow closer to the conceptual level of the original iTaSC workflow.

As a result, a summer intern was able to recreate the presented application in about two weeks, while the original implementation was developed with the aid of multiple students and PhD students, which together spent multiple person-months.

7.5 Conclusion

This chapter demonstrated a force-sensorless and bimanual human-robot comanipulation application, which integrates the wrench nulling control scheme of Chapter 6, the code support discussed in Chapter 4 and 5, and the lessons learned from Chapter 3 to 5. The task comprises 35 constraints in different control spaces for a 20 DOF, tree-structured robot. The task is implemented in a structured way using the constraint-based programming DSL, and instantiated using the iTaSC software framework. The sensorless wrench-nulling control scheme presented in Chapter 6, enabled direct human-robot interaction without the use of a force sensor. A video shows the performance of the full task with all described constraints in different scenarios. Quantitative results are provided for experiments involving a subset of constraints. The experiments validate the wrench-nulling, the joint-limits, and obstacle avoidance performance. Moreover, the experiments validate not only the functional aspects, but also the monitoring, coordination and configuration aspects of the different tasks. For example, the latter two tasks show the ability to activate and deactivate, as well as to change the weights of different constraints in a stable way.

Furthermore, the presented use case is a large application developed in the iTaSC software framework, which was developed with the aid of multiple students and PhD students, together spending multiple person-months. The resulting implementation was later re-developed by a summer intern, using the Composition Pattern and the resulting DSLs to design (model) the application, and using the iTaSC software framework to instantiate the application. It took the intern with basic knowledge of constraint-based programming about two

weeks to learn the software tools and recreate the application, a fraction of the original development time.

The iTaSC workflow still forms a good general guideline to create iTaSC applications, however focusses only on the functionality (computation). The presented DSL is more detailed, giving a template for all concerns.

The original code and the model of the recreated use case are publicly available on http://gitlab.mech.kuleuven.be/rob-itasc/itasc_comanipulation_demo.git, and http://bitbucket.org/dvanthienen/itasc_comanipulation_model, respectively.

The performance of the application can, however, still be improved in future work. The current implementation solves the task specification problem instantaneously which gives nervous behavior of the mobile base due to its non-holonomic character (it takes time to rotate the wheels in another direction). Furthermore, incorporating better joint velocity measurements could also improve the performance of the application. This enables the developer to remove the low-pass filters on these measurements, which could result in faster and more accurate behavior.

Chapter 8

Conclusions

This chapter summarizes the contributions of this dissertation in Section 8.1 and gives suggestions for future work in Section 8.2. It refers to the objectives numbered in Section 1.2.

8.1 Contributions

The two major categories of contributions of this dissertation are (i) the Composition Pattern to deal with the increasing complexity of robotic applications, and (ii) a force-sensorless wrench control scheme for service robots.

8.1.1 Force-sensorless and bimanual human-robot comanipulation

The sixth objective of this dissertation was to create a service robot application that integrates the different contributions of this dissertation into a single application.

The most prominent complex application developed in this dissertation is **force-sensorless and bimanual human-robot comanipulation**, discussed in Chapter 7. It formed the inspiration of, and the use case for, the other contributions of this dissertation, which are related to control and software. This use case shows **complex behavior emerging from the composition of constraints**. These constraints are expressed in multiple and different

generalized task spaces, including configuration, sensor, task, and feature spaces. For example, Section 7.3.2 explains and experimentally validates how following behavior emerges from the composition of weights on the different robot joints with joint limit and wrench nulling constraints. The latter applies the novel control scheme summarized in Section 8.1.2.

The developed use case was the first major application developed in the refactored iTaSC software framework. It was developed with the aid of multiple students and PhD students, together spending multiple man-months. The implementation was later **fast reprogrammed by a non-expert, using the Composition Pattern and the resulting DSLs**: a single summer intern with knowledge of constraint-based programming learned the software tools and recreated the application in about two weeks. The concrete **code was instantiated** using the iTaSC software framework.

The experiments of Section 7.3.2 validate not only the functional aspects, but also show the **performance of the support entities** introduced with the Composition Pattern, in particular the monitoring, coordination and configuration aspects of the different tasks.

The video of appendix C.1 shows the performance of the use case in a live demonstration.

The original **code and the model** using the DSLs developed in this dissertation have been made publicly available on `http://gitlab.mech.kuleuven.be/rob-itasc/itasc_comanipulation_demo.git` and `http://bitbucket.org/dvanthienen/itasc_comanipulation_model`, respectively.

8.1.2 Novel force-sensorless wrench control schemes in the resolved-velocity iTaSC approach and framework

The fifth objective of this dissertation was to develop a force-sensorless wrench control scheme for velocity controlled (service) robots. Chapter 6 presented three variations of a **novel force-sensorless wrench control scheme for velocity controlled robots:**

1. the wrench nulling control scheme shown in Figure 6.8;
2. the wrench control scheme shown in Figure 6.25, which expresses the force control task constraints in a six DOF Cartesian task space; and
3. the wrench control scheme shown in Figure 6.27, which expresses the force control task constraints in the robot joint space.

These control schemes are integrated in the resolved-velocity iTaSC approach and software framework.

The first control scheme is successfully applied to force-sensorless and bi-manual human-robot comanipulation explained in Section 6.3 and Chapter 7. In this application, two wrench nulling tasks, each defined between a gripper of the robot and its base, result in human-following behavior of the robot. The pushing and table wiping experiments of Section 6.7 validate the latter two control schemes, and show their applicability to service robot tasks. They show that the robot is able to deliver a **stable, constant contact wrench**. The wrench applied by the robot is highly **repeatable** in a certain contact point, however its accuracy depends on the robot configuration.

The experiments result in a set of concrete **guidelines to maximize the performance** of the presented control schemes, summarized in Section 6.9. The worst-case measurement error in a vertical direction, when applying the guidelines, is $0.47N$ or about 10% of the desired value. This **level of accuracy fits service robot tasks** such as table wiping or screwing a screw in wood.

The presented control schemes are applicable to velocity controlled robots for which joint velocity errors reflect torque disturbances. More specifically, it can be applied to a robot (i) with backdrivable robot joints, (ii) with known, only proportional gains for the lower-level joint velocity control loops, and (iii) with sufficiently high-frequency lower-level controllers, allowing their interactions to be neglected.

These control schemes have following **advantages** over alternative approaches, such as impedance or hybrid force/motion control:

- The control schemes **do not require a precise dynamic model of the robot, environment or contact point**.
- The control schemes allow the **combination of force control task constraints with other task constraints** in the resolved-velocity iTaSC approach and framework.
- The control schemes **avoid an expensive and complex force sensor**, which is not always present on service robot platforms such as the PR2 robot.
- The control schemes feature a reference adaptation factor, which can be applied to **impose desired transient behavior** on the applied force.

A software **implementation** of the presented control schemes, integrated in the iTaSC software framework, and including support for code instantiation

from the DSLs of Chapter 4, can be found in: https://bitbucket.org/dvanthienen/force_control_dsl.

8.1.3 The Composition Pattern as a systematic approach to robot application development

The first objective of this dissertation was to develop a systematic approach to create more flexible, robust, reusable, and adaptable robot applications. This dissertation introduced the Composition Pattern and the systematic procedure to apply the Composition Pattern to the modeling of robot applications as a **uniform and easy-to-grasp way to deal with complexity, from software architecture to behavior composition**. The Composition Pattern builds on the metamodeling concept, which considers **all entities to be models**, not objects. It defines a **Composite Functional Entity as first-class citizen of system design**. This Composite Functional Entity consists of one Coordinator, one Composer, one Scheduler, one or more Configurators, one or more Monitors, one or more Communicators, and one or more Functional Entities, as shown in Figure 3.3. Each Functional Entity can be replaced by a Composite Functional Entity –referred to as a 'composite' in short– forming a hierarchy of entities as shown in Figures 3.4 and 5.1.

The **hierarchy** in this dissertation replaces strict interactions, e.g. used in programming to the interface, by soft constraints. Moreover, it allows some functionality to cross the hierarchical boundaries and it gives composite functional entities some **local responsibility** through the support entities. This results in improved system performance due to short communication paths and fast reactions, in contrast to 'everything needs to go through upper management', as exemplified by the pick-and-place example in Box 3.3.4.

In addition, an entity has a boundary, defined as a **semantic context**. It implies a shared vocabulary between the entities within a composite. Section 3.4 shows how this results in a more flexible, robust, and reusable system, by forming a **boundary** to what the support entities of a composite have to manage, and by **decoupling** functional entities from their support entities. However, it is a *soft* boundary which does **not imply information hiding**, hence enables introspection or reasoning.

The advantages of the Composition Pattern are:

- It aids developers to **cope with the increasing scale and complexity** of robotic applications.

- It provides a systematic approach to **create and refactor robust, flexible, reusable, and adaptable software**.
- It **can be applied to the developers' framework ecosystem of choice**.
- It provides a concrete and **constructive way to apply the 5Cs principle of separation of concerns**.

These advantages are made explicit in Section 7.4, which shows how the Composition Pattern made it easy to recreate the complex use case of human-robot comanipulation by modeling the application using the presented DSL. Moreover, the Composition Pattern is thaught to students as part of the Advanced Robotics and Control Systems course and Embedded Control Systems course at KU Leuven, and forms the approach applied in new software project developments.

The Composition Pattern forms the most conceptual contribution of this dissertation, the more concrete results of the Composition Pattern are formalized as separate contributions in following sections. These contributions limit themselves to the application of the Composition Pattern to task specification and control, hence this dissertation does not provide the final answer on how to best apply the methodology to any new application. Concretely, the Composition Pattern is used to structure and formalize constraint-based programming in a domain-specific language (DSL), which will be summarized as separate contributions in Section 8.1.4 and 8.1.5. Moreover, the Composition Pattern is applied as an architectural pattern to the iTaSC software framework, which will be summarized as a separate contribution in Section 8.1.6.

8.1.4 Modelling of the domain of constraint-based programming using the Composition Pattern

The second objective of this dissertation was to model the domain of constraint-based programming using the Composition Pattern. Modeling of software forces developers to think about what would be needed to let non-software (but domain) experts exploit software frameworks. Therefore, it has become the main driver in our research group for **structured coding**. Moreover, models are an important requirement to give a robot the **possibility to reason, on- or off-line, on its own functionality**, a first step to cognitive robot systems.

Modeling the domain of constraint-based programming provides a **generic division of the domain**. It divides the complexity of a constraint-based

programming application in a hierarchical structure of about 17 separate concepts (entities), shown in Figure 3.5. This structure forms the blue print for

- the formulation of the domain meta-model using a domain-specific language (DSL), summarized in following section, and
- the refactoring of the iTaSC software framework, summarized in Section 8.1.6.

8.1.5 Formulation of constraint-based programming meta-model using a domain-specific language (DSL)

This dissertation formulated the meta-model of the domain of constraint-based (robot) programming using a domain-specific language (DSL) achieving the third objective. This DSL brings the **systematic way** of describing an iTaSC application of previous research [46] (the iTaSC workflow) to a higher level. The structure of this DSL is outlined in Figure 4.3, a reference example model of this DSL is given in Section 4.5. However, the provided DSL does not describe all sub-domains of a constraint-based programming application. It integrates more specific DSLs, for example it integrates rFSM as meta-model of Coordination entities. As such, the DSL forms the **starting point for the integration of existing and future modeling efforts**, in particular future modeling efforts on Composition, Scheduling, Monitoring, and Communication.

The overall effort of creating an application using the metamodeling approach is larger than hand-coding a single application from scratch. However, using the metamodeling approach reduces the amount of hand-written code and the development time, when creating or adapting applications in the same domain. Concretely, the DSL enables developers or users to **easier (and hence faster) understand and (re)-program constraint-based programming applications**, using the DSL as a template. For example, the recreation of the human-robot comanipulation use case detailed in Chapter 7 took a summer intern about two weeks to grasp the concept and create the application, while the original application took experts man-months to create.

Moreover, the DSL enables **automatic model verification**, and manual or automatic **code generation** to the software framework ecosystem of choice. As shown in Section 4.7, automatic model verification returns meaningful errors, effectively reducing code debugging efforts. The model to meta-model conformity can be verified using the uMF software tooling [92].

The implementation of the DSL, as well as the **software tool** to transform the concrete models into an implementation using the iTaSC software framework, are made publicly available under an open-source software licence. The implementation of the DSL can be found on http://bitbucket.org/dvanthienen/itasc_dsl. The software tool to transform a model into an implementation can be found on http://bitbucket.org/dvanthienen/itasc_dsl_orocos_deployer.git

8.1.6 The Composition Pattern as an architectural pattern and its application to iTaSC

The fourth objective of this dissertation is to develop a flexible, robust, and reusable software framework for constraint-based programming. This dissertation showed how the Composition Pattern can be used as a software **architectural pattern**, by applying it to the iTaSC software framework, a generalized constraint-based programming approach [46]. It serves as a primary example of the application of the Composition Pattern to the broad system integration context of a robotics system, since (i) task specification, execution and monitoring involves 'planning', 'sensing', 'control', and 'world modeling' functionalities, and (ii) it was the framework in which we first encountered the fundamental deficiencies of former design 'guidelines'.

The application to the iTaSC software framework learned following lessons:

- Separation of concerns is often used in isolation, i.e. 'separation of concerns hence reusable entities'. We learned that **composition**, which forms the fifth C of the 5Cs approach to separation of concerns, **is as important as separation**. The Composition Pattern provides a constructive way to apply these 5Cs.
- This dissertation started from component based frameworks such as Orocos [33] and ROS [171], believing that *'components'* are the fundamental building blocks for reusability of functionalities in various architectural compositions. Now, the **Composition Pattern** shown in Figure 5.1 has become the **first-class citizen in our system design**. Components are still necessary, but are not fundamental building blocks anymore. This difference is important, since a component that is *designed* to be part of the Composition Pattern will be different from one designed without that context.
- The Composition Pattern introduces an elaborate structure. The advantage however is the **reduced configuration files or software**

libraries, because developers find it a lot easier to define the scope of each particular software development effort.

The **reference implementation of the iTaSC software framework** uses, in itself, two other large-scale software frameworks, *Orocos* [33] and rFSM [91]. It is made publicly available as a set of repositories: The core functionality and template components for each entity type can be found in

- http://gitlab.mech.kuleuven.be/rob-itasc/itasc.git, and
- http://gitlab.mech.kuleuven.be/rob-itasc/itasc_core.git.

Libraries of different components for each entity type can be found in

- http://gitlab.mech.kuleuven.be/rob-itasc/itasc_robots_objects.git,
- http://gitlab.mech.kuleuven.be/rob-itasc/itasc_solvers.git,
- http://gitlab.mech.kuleuven.be/rob-itasc/itasc_tasks.git,
- http://gitlab.mech.kuleuven.be/rob-itasc/itasc_constraint_controllers.git,
- http://gitlab.mech.kuleuven.be/rob-itasc/itasc_virtual_kinematic_chains.git, and
- http://gitlab.mech.kuleuven.be/rob-itasc/itasc_examples.git.

8.2 Future work

Future work related to force-sensorless wrench control

This dissertation validated the novel force-sensorless wrench control schemes on a PR2 robot. Future work should validate the control scheme on a variety of robot platforms to analyse the platform dependency of observed effects. In particular, it should analyse the effect of the increased accuracy in the applied

force when following a procedure of (i) first applying a force setpoint a little higher than desired, and then (ii) decreasing it to the desired force setpoint. Furthermore, future work should research whether the performance on the PR2 platform can be increased

- by including gravity compensation in the robot wrist joints, or
- by adding an additional, configuration-dependent force feedforward term to compensate for the experimentally observed force offsets.

Future work related to the Composition Pattern and its application to constraint-based programming

The application of the Composition Pattern to constraint-based programming, the development of a DSL, and the refactoring of the supporting software framework, showed the possibility –but also need– to integrate more models and tools.

The current implementation presents only a limited set of DSLs, future work should focus on the integration of dedicated DSLs for all entities. Also the number of entities can be enlarged, for example, by expanding the scope to composites of applications, i.e. *missions*. Furthermore, this integration and broadening should also be applied to other frameworks. For example, future work could provide tools to instantiate the presented models on other constraint-based programming frameworks such as Stack of Tasks [102]. Another interesting example of future work is to translate iTaSC DSL models to the recently developed expressiongraph-based task specification language (eTaSL) [3]. Integration with eTaSL allows developers to extend task specifications supported by the iTaSC software framework, which are limited to six DOF Virtual Kinematic Chains and position loop closure constraints, to more general task expressions, including inequality constraints. A limitation of the current implementation is that the current tooling allows only a 'static' and 'one way' instantiation of models. Future work should research and develop tools to make this model-implementation relation dynamic, reflecting run-time changes in the models.

Moreover, the development time and ease could be improved by providing tool support for design time model checking, using for example Xtext [56]; or by integrating the presented DSL and workflow in graphical programming tools, such as ABB's RobotStudio [1]. Another, practical consequence of the modeling of the domain in separate, reusable entities is the possibility for the creation of a repository or 'store' with models and/or implementations of entities, for example a 'task store' for robot task models.

In the long run, future work should determine how to best apply the Composition Pattern to any new application. It should research how the Composition Pattern can be applied to perception, planning, etc. The applicability of the Composition Pattern is however limited, it does not provide a methodology to create, for example, new algorithms, class libraries, or schedulers, although it gives these concepts a place in system development.

This dissertation did not research the full potential of the proposed approach. Important will be (i) the integration of reasoning, on all composites, on all tiers; and (ii) the development of a larger tool set for formal verification and validation.

However, this requires a lot more work to provide a broader set of structural and behavioral *models* within the robotics community, and the necessary development of *tooling* to help developers at creating applications, beyond the examples given above. A broad and consistent application of the Composition Pattern could be a significant driver to accelerate these developments. Additionally, formal modelling paves the way for robots to generate their own behavior, and reason on their behavior on a symbolic level.

Appendix A

Proofs

A.1 Proof of invertibility of B

This section proofs that

$$\boldsymbol{B} = \boldsymbol{I} - \boldsymbol{A}_W^{\#} c_f \boldsymbol{S}^{\#} \boldsymbol{J}_{\boldsymbol{q}}, \tag{A.1}$$

defined in Section 6.5, is invertible.

For a matrix to be full rank or non-singular, its determinant should be non-zero. Hence for $\boldsymbol{B}$ to be of full rank, the determinant of $\boldsymbol{B}$ should not be equal to zero

$$det(\boldsymbol{B}) = det(\boldsymbol{I} - \boldsymbol{A}_W^{\#} c_f \boldsymbol{S}^{\#} \boldsymbol{J}_{\boldsymbol{q}}) \neq 0. \tag{A.2}$$

Using *Sylvesters determinant theorem*, stating that $det(I_p - XY) = det(I_n - YX)$ for a matrix X of size $(p \times n)$ and a matrix Y of size $(n \times p)$, equation (A.2) can be reordered to

$$det(\boldsymbol{B}) = det(\boldsymbol{I} - \boldsymbol{S}^{\#} \boldsymbol{J}_{\boldsymbol{q}} \boldsymbol{A}_W^{\#} c_f) \neq 0. \tag{A.3}$$

Assuming no conflicting constraints in the force controlled directions, equation (A.3) reduces to

$$det(\boldsymbol{B}) = det(\boldsymbol{I} - c_f) \neq 0. \tag{A.4}$$

Given that c_f is a $(n_{sel} \times n_{sel})$ diagonal matrix, equation (A.4) holds if $c_f \neq I$.

Appendix B

Additional experimental data of force-sensorless force control

This chapter discusses additional experimental data of the force-sensorless force control schemes presented in Chapter 6.

B.1 Analysis of the force control task constraints in Cartesian task space

This section analyses the force control task constraints in Cartesian task space. These constraints express a desired twist or part thereof. This desired twist is function of two variables that change during the task execution: the robot Jacobian $\boldsymbol{J}_{\boldsymbol{q}}$, and the velocity error $\boldsymbol{e}_{\dot{\boldsymbol{q}}}$. In case the robot reaches equilibrium, hence steady-state, both these variables are constant, the latter zero. Therefore, the figures give insight in the transient behavior.

Each of the figures discussed in this section contains the desired twist for all repetitions of the experiments, in all contact points, hence thirty experiments in total. $K_a = 0.1$ for all force or torque controlled directions in all experiments.

Figure B.1 shows the desired velocity $\dot{\boldsymbol{y}}_{d,1}^{\circ} = \dot{z}_d$ for the case when the robot applies a force in the z-direction with $K_a = 0.1$, explained in Section 6.7.3

Figure 6.35 shows the corresponding wrench measurements. The figure shows that the trajectories overlap for the different repetitions in each contact point, showing the repeatability of the experiments. The transitions between desired velocities have little overshoot, as expected for the chosen value of the reference adaptation factor K_a. The rise time takes less than half a second, with exception of the initial contact, which takes $1.5s$.

Figure B.2 shows the desired twist $\dot{\boldsymbol{y}}_{d,1}^{\circ} = \mathbf{t}_d^{\circ}$ for the case when the robot applies a force in the z-direction, while nulling the forces and torques in the other task space directions. K_a equals 0.1 for all force or torque controlled directions. Section 6.7.4 analyses the experiments of this case. Figure 6.41 shows the corresponding wrench measurements. The figure shows little more variation in applied velocities between different repetitions of the experiment in the same point. Different arm configurations and contact points could explain this variation.

Figure B.3 shows the desired velocities $\dot{\boldsymbol{y}}_{d,1}^{\circ} = \begin{bmatrix} \dot{y}_d \\ \dot{z}_d \end{bmatrix}$ for the case when the robot applies a force in the y- and z-direction, explained in Section 6.7.4. Figure 6.43 shows the corresponding wrench measurements. K_a equals 0.1 for both the y- and z-direction. Also this figure shows little more variation in applied velocities between different repetitions of the experiment in the same point. Different arm configurations or contact points could explain this variation. Moreover, the figure shows overshoot on the transients and a slightly higher rise time with respect to Figure B.1. Experiments in the z-direction where used to choose the reference adaptation factor K_a used in all directions, this experiments should be retaken in the y-direction to choose an apt factor in this y-direction. The desired velocity does not show any drift, although the drift is present in the measured force.

B.2 Force in z expressed as constraints in joint task space and without reference adaptation

Figure B.4 shows the measured force over time of the experiments applying a force in the z-direction with constraints expressed in joint task space. No reference adaptation is applied, hence $K_a = 0$. Table B.1 shows the average of the measurement standard deviations over the ten repetitions of a measurement in a certain contact point. Figure B.5 shows the distribution of the measurement averages for the different steps in applied force.

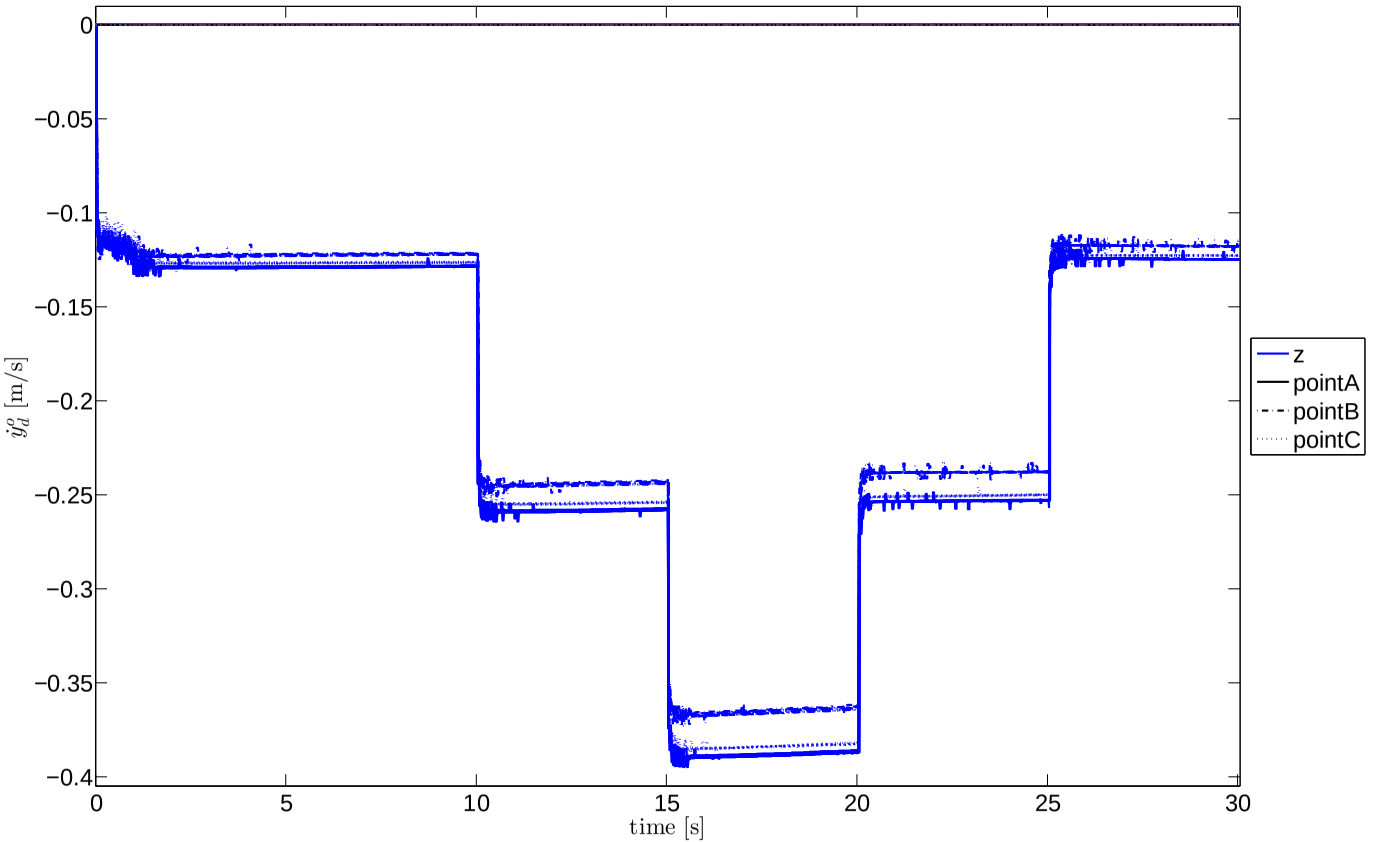


Figure B.1: Desired velocity over time for the case when the robot applies a force in the z-direction with $K_a = 0.1$. The experiments are repeated ten times, in three different contact points. Different line styles differentiate the contact points.

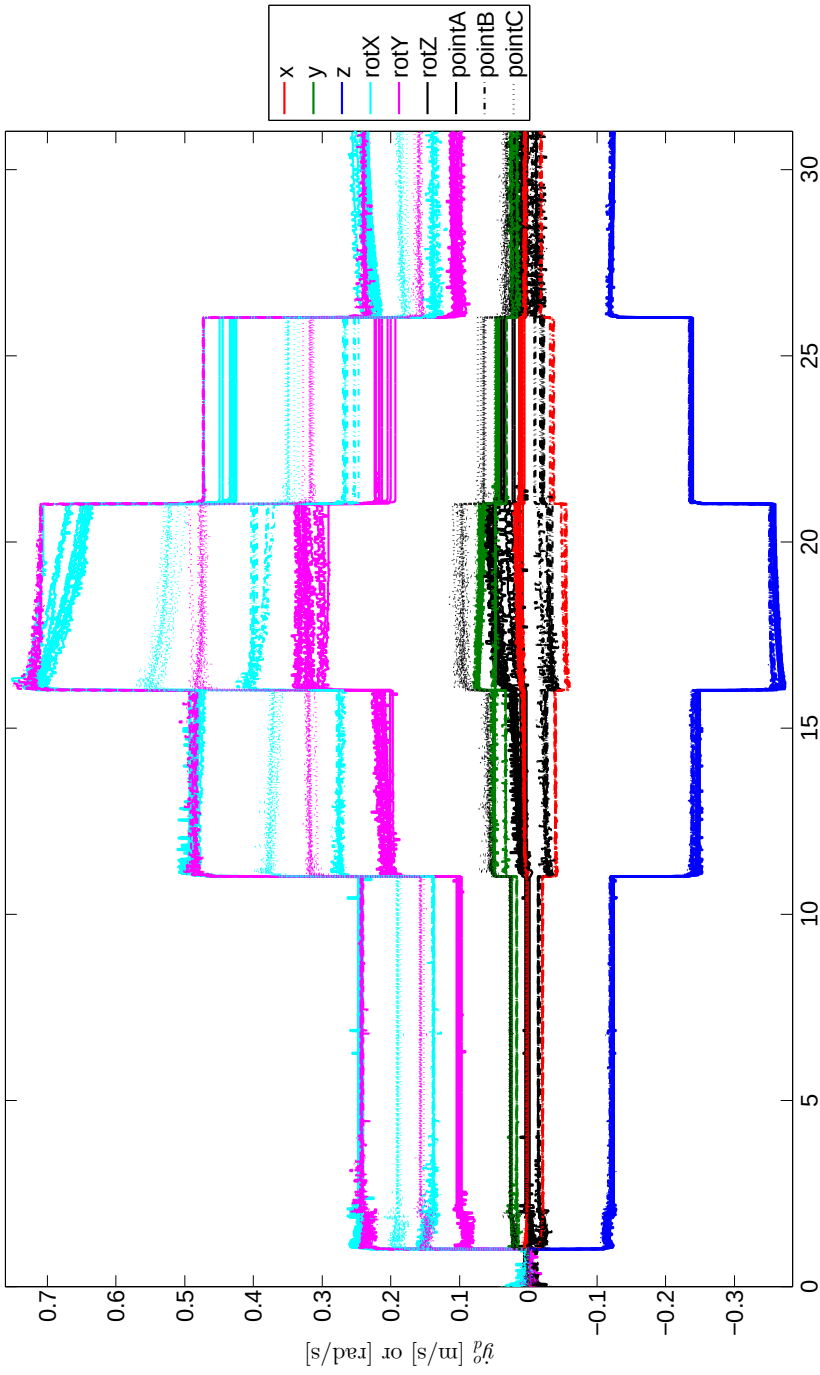


Figure B.2: Desired twist over time for the case when the robot applies a force in the z-direction, while nulling the forces and torques in the other task space directions. K_a equals 0.1 for all force or torque controlled directions. The experiments are repeated ten times, in three different contact points. Different line styles differentiate the contact points.

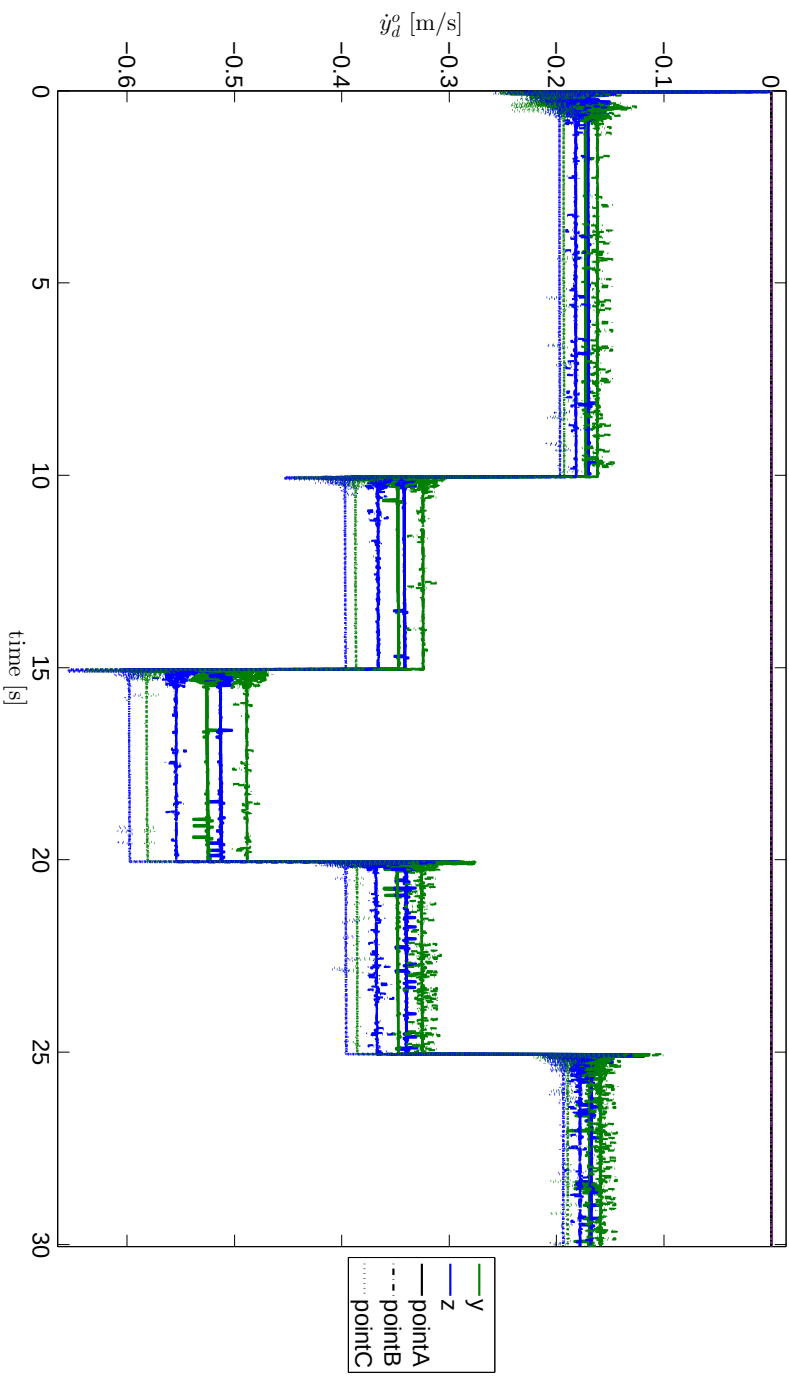


Figure B.3: Desired velocities over time for the case when the robot applies a force in the y- and z-direction. K_a equals 0.1 for both the y- and z-direction. The experiments are repeated ten times, in three different contact points. Different line styles differentiate the contact points.

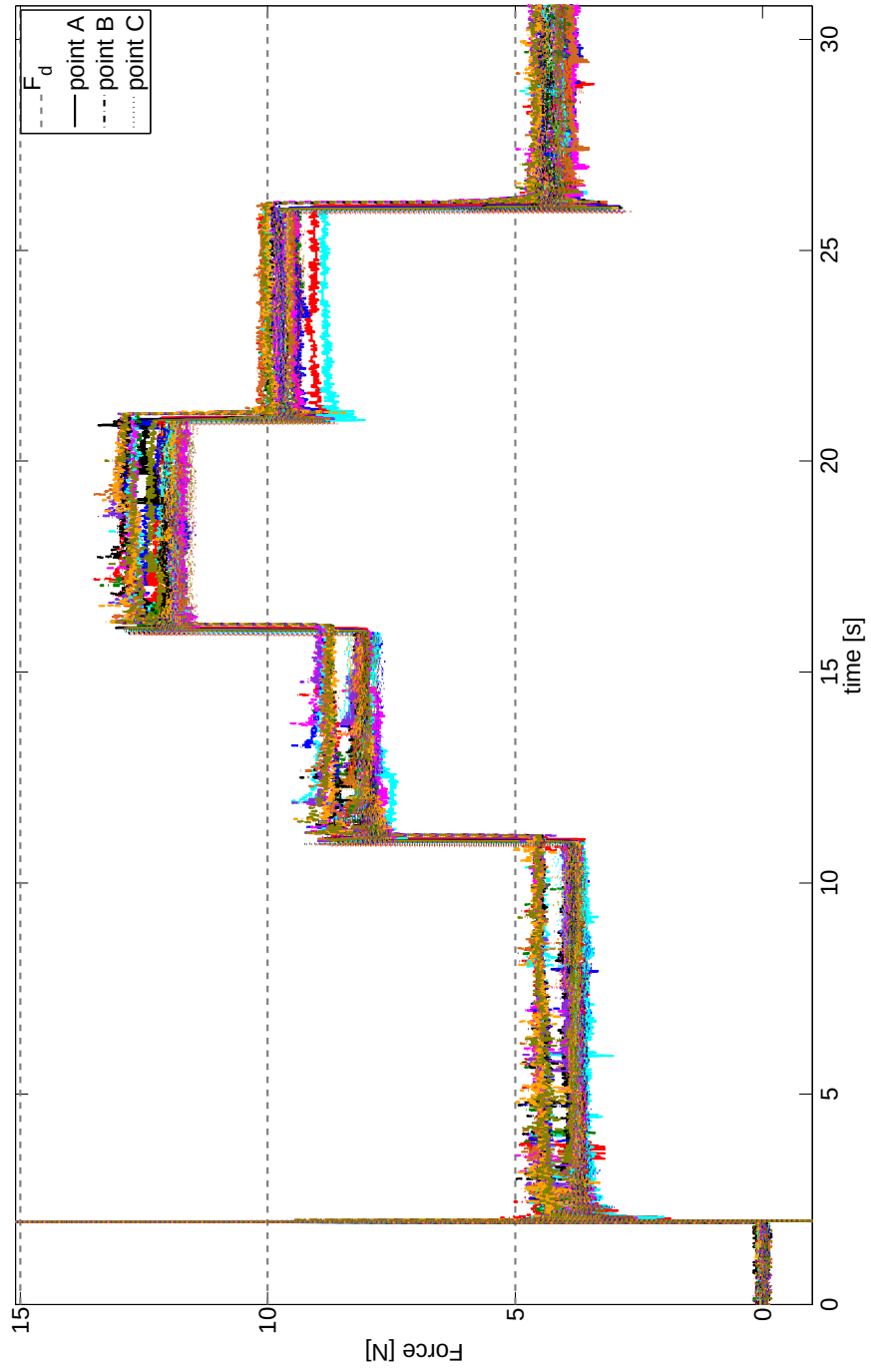


Figure B.4: Measured force over time of the experiments applying a force in the z-direction with constraints expressed in joint task space. No reference adaptation is applied. Grey, dashed lines indicate the desired values. The experiments are repeated ten times, in three different contact points. Different line styles differentiate the contact points, different colors differentiate experiments. The large spike shows the excitation of the force sensor dynamics when the gripper makes contact with it.

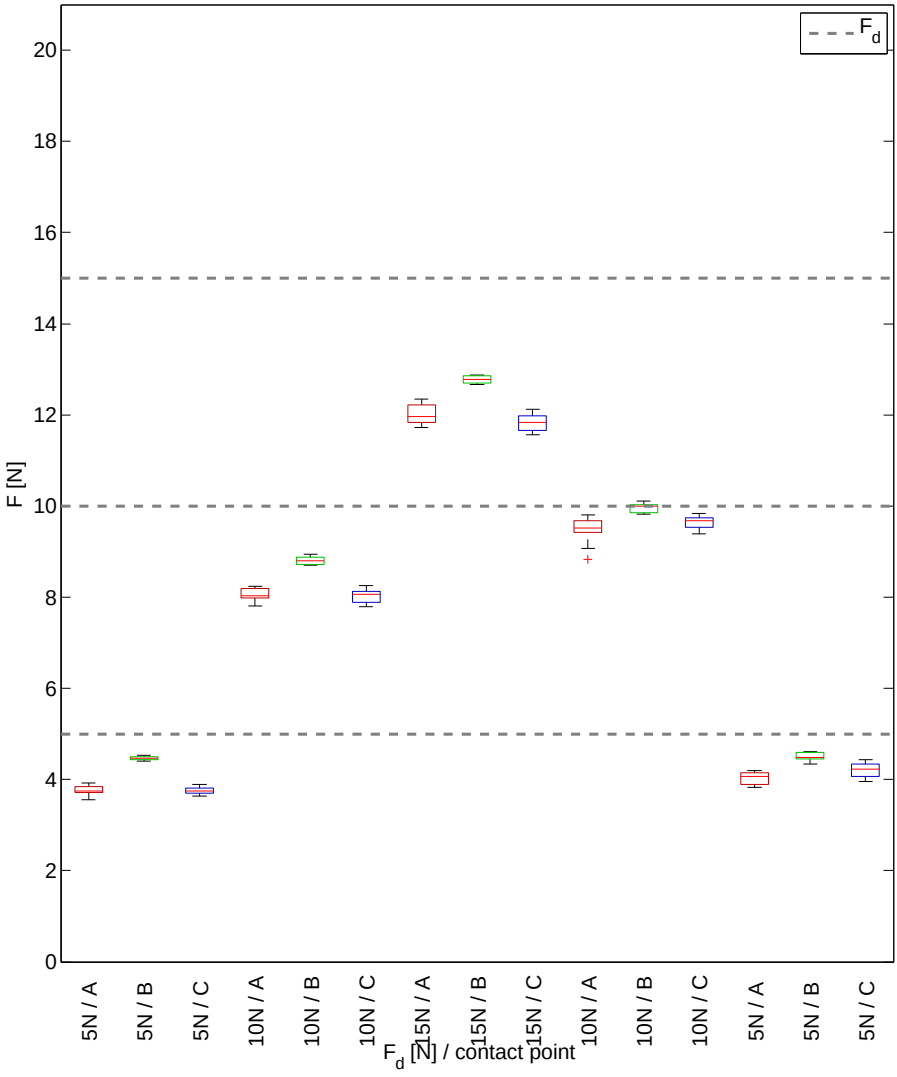


Figure B.5: Boxplot of the steady-state averages of the experiments applying a force in the z-direction with constraints expressed in joint task space. No reference adaptation is applied. Grey, dashed lines indicate the desired values. The measured force over time is shown in Figure B.4.

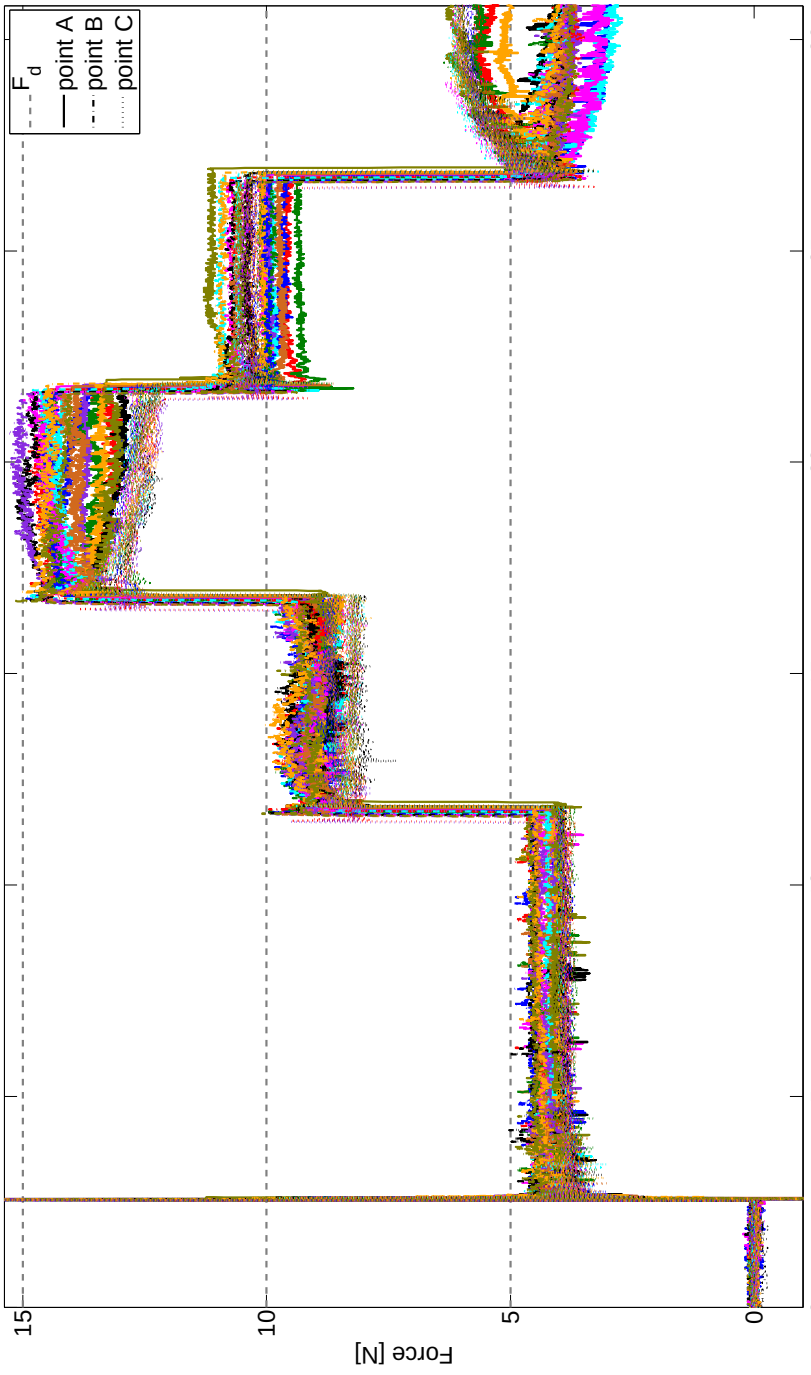


Figure B.6: Measured force over time of the experiments applying a force [0 0 x 0 0 0] with constraints expressed in Cartesian task space. No reference adaptation is applied. Grey, dashed lines indicate the desired values. The experiments are repeated ten times, in three different contact points. Different line styles differentiate the contact points, different colors differentiate experiments. The large spike shows the excitation of the force sensor dynamics when the gripper makes contact with it.

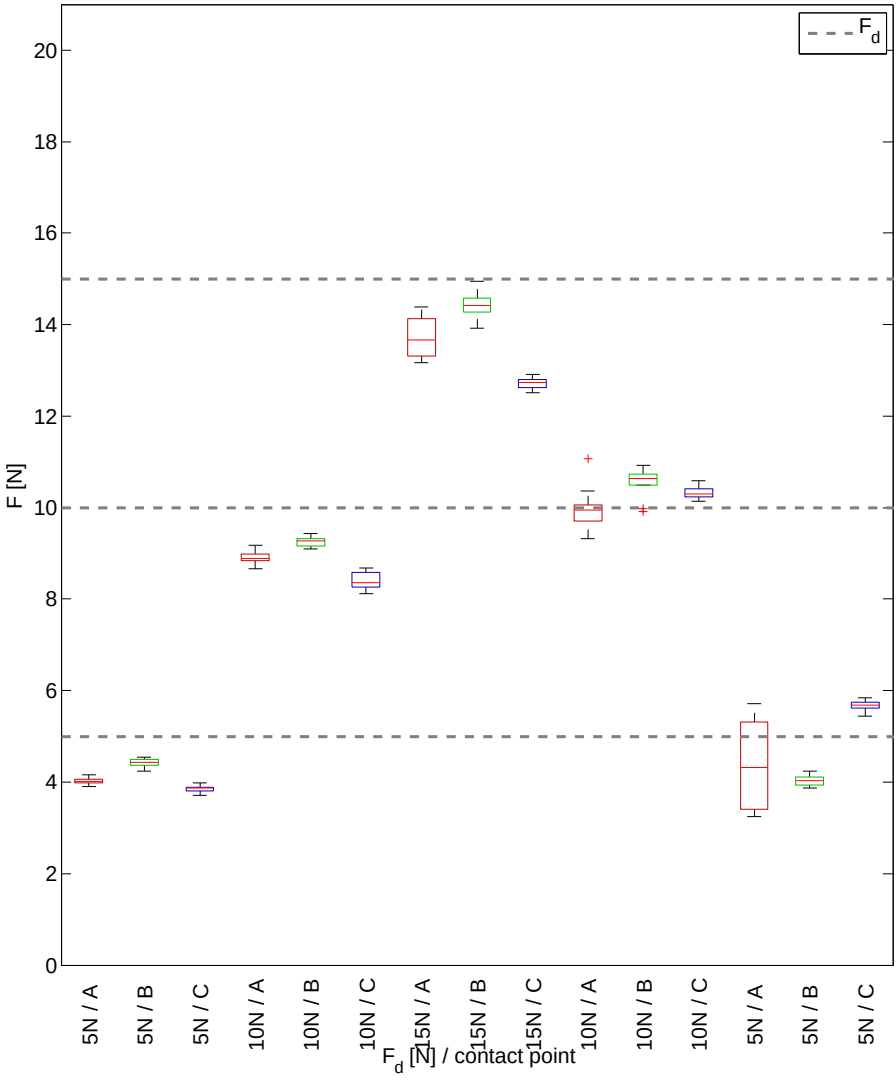


Figure B.7: Boxplot of the steady-state averages of the z-direction of the experiments expressing the force control task constraints in joint space. Grey, dashed lines indicate the desired values. The measured force over time is shown in Figure 6.41.

F_d	5N	10N	15N	10N	5N
point A	0.08	0.10	0.11	0.05	0.06
point B	0.11	0.10	0.14	0.06	0.08
point C	0.10	0.11	0.06	0.06	0.07
total	0.10	0.11	0.11	0.06	0.07

Table B.1: Average of the measurement standard deviations, expressed in Newton.

B.3 Force in the z-direction, explicitly specifying the force in other directions as zero and without reference adaptation

Figure 6.41 shows the measured force over time of the experiments applying a force [0 0 x 0 0 0] with constraints expressed in Cartesian task space. No reference adaptation is applied, hence $K_a = 0$. Table B.2 shows the average of the measurement standard deviations over the ten repetitions of a measurement in a certain contact point. Figure B.7 shows the distribution of the measurement averages for the different steps in applied force.

F_d	5N	10N	15N	10N	5N
point A	0.10	0.16	0.15	0.08	0.30
point B	0.10	0.20	0.15	0.06	0.18
point C	0.09	0.13	0.23	0.06	0.36
total	0.10	0.16	0.18	0.07	0.28

Table B.2: Average of the measurement standard deviations, expressed in Newton.

B.4 Table wiping use-case

This section details the experimental data of the table wiping experiments explained in Section 6.7.5. The experiment is repeated around two different points on the table, with for each two different Lissajous figures to trace. Each of the Lissajous figures is traced with two different speeds, characterized by the period to complete a figure. The experiment in the first point with the smallest period is repeated three times in order to study repeatability. Figure 6.29 indicates the location of the two different contact points on the table, D and

E, used in the table wiping experiments. Figures B.8 and B.9 show the figures traced by the robot gripper on the table, and their respective desired figures.

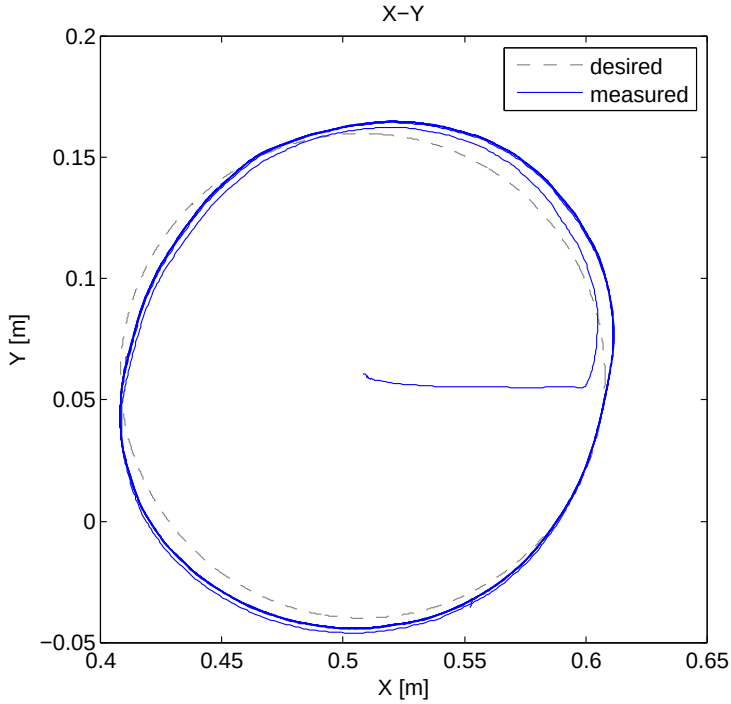


Figure B.8: Circle traced by the robot gripper on the table for the first experiment in point E with a period of 20s.

Table B.3 and B.4 show the average and standard deviation of the measured force on the table for the different experiments, respectively. The sensor is nulled between experiments. However due to the setup, the measurement is disturbed by the sensor and wiper mount. Therefore, the force averages are offset with respect to the real applied average force, and can only be compared relatively.

figure	circle				flower			
period	20s			10s	40s			20s
point D	5.99	5.43	6.11	5.42	5.43	5.63	5.51	5.59
point E	5.47			5.33	5.39			5.97

Table B.3: Average of the measurements in Newtons

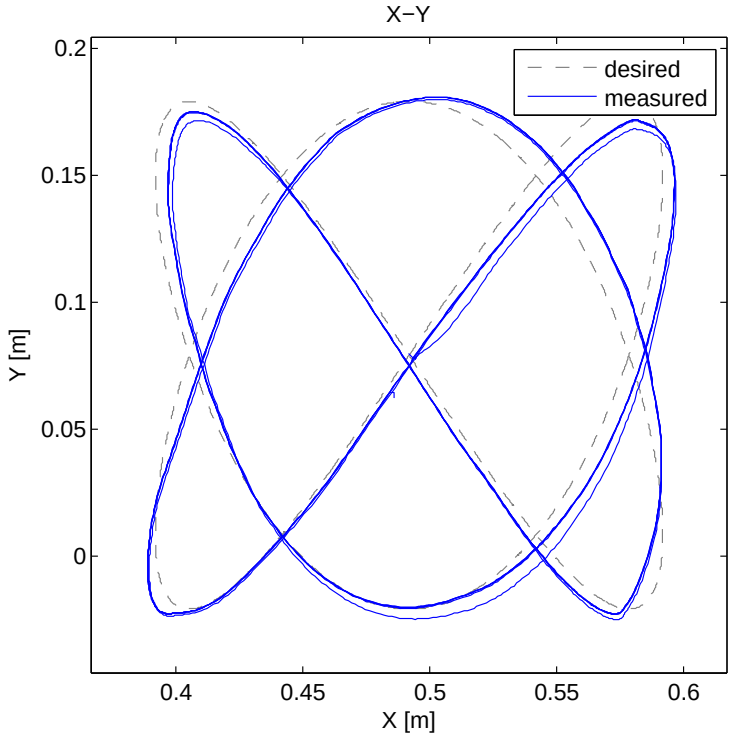


Figure B.9: Flower traced by the robot gripper on the table for the first experiment in point E with a period of 20s.

figure	circle				flower			
period	20s			10s	40s			20s
point D	0.77	0.85	0.87	0.95	0.94	0.94	1.16	1.18
point E	0.92		1.15	1.21		1.15		

Table B.4: Standard deviation of the measurements in Newton

Appendix C

Videos

This chapter lists and explains the videos referenced in this dissertation. The videos are made available online [164] on

http://people.mech.kuleuven.be/~dvanthienen/thesis/videos.html

C.1 Force-sensorless human-robot comanipulation

Referenced in Sections 7.3, 7.5 and 8.1.1.

This video shows a human-robot comanipulation application, where a robot helps a human carrying a plate. The robot follows the directions of the human by reacting on the wrenches applied on the plate. The human directs the robot through a door and back, while the robot avoids hitting the door (obstacle avoidance). A detailed explanation can be found in Chapter 7.

C.2 Applications developed using the constraint-based programming DSL

C.2.1 Drawer opening using a PR2 robot

Referenced in Section 4.3.

The video shows a drawer opening application with a PR2 robot. In this application, the robot has to (i) reach for the handle with its right gripper, (ii) grasp the handle, and (iii) open the drawer, (iv) while keeping close to a preferable joint configuration, and (v) staying away from joint limits.

C.2.2 Variations on a Lissajous tracing task

Referenced in Section 4.7.

The videos show the execution of a laser tracing application on three different platforms, i.e. a KUKA LWR, a KUKA YouBot, and a Willow Garage PR2, and with respect to three different objects, i.e. a table, the floor, and a wall. The application defines a task to trace a Lissajous figure on the surface of an object. This laser tracing task is defined between an end-effector of the robot and an object. A model written in the constraint-based programming DSL models the application. This model is adapted to execute the task on another platform or with respect to another object. In addition, platform specific tasks are adapted to each platform change, e.g. joint limit avoidance.

C.3 Force-sensorless force control

C.3.1 Force in z expressed as constraints in Cartesian task space

Referenced in Section 6.7.3.

This video shows the PR2 robot applying a force on a sensor on a table in point A[†], as explained in Section 6.7.3. The robot arms moves slightly when increasing the desired force.

C.3.2 Force in y expressed as constraints in Cartesian task space

Referenced in Section 6.7.3.

These videos show, from two viewpoints, the PR2 robot applying a force on a sensor, mounted on the side of a heavy structure in point A[†]. Section 6.7.3

[†]Section 6.7.1 defines the experimental setup and contact points.

analyses the results of the experiment. The robot arms moves slightly when increasing the desired force.

C.3.3 Applying a force in the Cartesian y- and z-direction

Referenced in Section 6.7.4.

This video shows the PR2 robot applying a force on a sensor on a table in point C[†], as explained in Section 6.7.4. When changing the desired force, the robot arm moves and also the contact point changes.

C.3.4 Force in the z-direction, explicitly specifying the force in other directions as zero

Referenced in Section 6.7.4.

This video shows the PR2 robot applying a force on a table, while the robot maintains a zero force or torque in the other (Cartesian) directions. The desired force applied to the system is first 5N, then increased to 10N and 15N, while maintaining contact. Then the desired force is decreased to 10N and 5N. Since there is no pose controller to enforce position constraints, the robot arm moves the contact point when the force setpoint is altered. In case of the 15N and the decreased 5N desired force, the robot moves in the negative and positive y-direction, respectively, before reaching equilibrium.

C.3.5 Table wiping use-case

Referenced in Section 6.7.5.

These videos show the PR2 robot wiping a table around point D[†], as explained in Section 6.7.5. The different videos show variations on the experiment, where the robot traces a flower or a circle, slow or fast.

[†]Section 6.7.1 defines the experimental setup and contact points.

Bibliography

[1] ABB. ABB Robotics. http://www.abb.com/robotics/, 2011.

[2] Abdellatif, T., Bensalem, S., Combaz, J., de Silva, L., and Ingrand, F. Rigorous design of robot software: A formal component-based approach. *Robotics and Autonomous Systems 60*, 12 (2012), 1563–1578.

[3] Aertbeliën, E., and De Schutter, J. eTaSL/eTC: A constraint-based task specification language and robot controller using expression graphs. In *Proceedings of the 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Chicago, IL, USA, 2014), IROS2014, pp. 1540–1546.

[4] Alami, R., Chatila, R., Fleury, R., Ghallab, M., and Ingrand, F. An architecture for autonomy. *The International Journal of Robotics Research 17*, 4 (1998), 315–337.

[5] Anderson, R. J., and Spong, M. W. Hybrid impedance control of robotics manipulators. *IEEE Journal of Robotics and Automation 4*, 5 (1988), 549–556.

[6] Ando, N., Suehiro, T., Kitagaki, K., Kotoku, T., and Yoon, W. K. RT-Middleware: distributed component middleware for RT (robot technology). In *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Edmonton, Canada, 2005), IROS2005, pp. 3933–3938.

[7] Ando, N., Suehiro, T., and Kotoku, T. A software platform for component based RT-system development: OpenRTM-Aist. In *International Conference on Simulation, Modeling, and Programming of Autonomous Robots* (Venice, Italia, 2008), pp. 87–98.

[8] Angerer, A., Hoffmann, A., Schierl, A., Vistein, M., and Reif, W. Robotics API: Object-oriented software development for industrial robots. *Journal of Software Engineering in Robotics 4*, 1 (2013), 1–22.

[9] Baerlocher, P. *Inverse kinematics techniques of the interactive posture control of articulated figures*. PhD thesis, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2001.

[10] Baerlocher, P., and Boulic, R. Task-priority formulations for the kinematic control of highly redundant articulated structures. In *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Vancouver, British Columbia, Canada, 1998), IROS98, pp. 323–329.

[11] Baeten, J., Bruyninckx, H., and De Schutter, J. Integrated vision/force robotics servoing in the task frame formalism. *The International Journal of Robotics Research 22*, 10 (2003), 941–954.

[12] Bálek, D., and Plášil, F. Software connectors and their role in component deployment. In *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems (DAIS)* (2001), pp. 69–84.

[13] Barnes, M., and Finch, E. L. COLLADA—Digital Asset Schema Release 1.5.0. http://www.collada.org, 2008. Last visited January 2015.

[14] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*, 2 ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[15] Basu, A., Bozga, M., and Sifakis, J. Modeling heterogeneous real-time components in bip. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods* (Washington, DC, USA, 2006), SEFM '06, IEEE Computer Society, pp. 3–12.

[16] Beetz, M., Mösenlechner, L., and Tenorth, M. CRAM—A cognitive robot abstract machine for everyday manipulation in human environments. In *Proceedings of the 2010 International Conference on Advanced Robotics* (2010), pp. 1012–1017.

[17] Ben-Israel, A., and Greville, T. N. E. *Generalized Inverses: Theory and Applications*, reprinted ed. Robert E. Krieger Publishing Company, Huntington, NY, 1980.

[18] Bensalem, S., de Silva, L., Ingrand, F., and Yan, R. A verifiable and correct-by-construction controller for robot functional levels. *Journal of Software Engineering in Robotics 2*, 1 (2011), 1–19.

[19] Bensalem, S., Gallien, M., Ingrand, F., Kahloul, I., and Nguyen, T.-H. Toward a more dependable software architecture for autonomous robots. *IEEE Robotics and Automation Magazine 16* (2009).

[20] Bézivin, J. On the unification power of models. *Software and Systems Modeling 4*, 2 (2005), 171–188.

[21] Bohren, J., Rusu, R. B., Jones, E. G., Marder-Eppstein, E., Pantofaru, C., Wise, M., Mosenlechner, L., Meeussen, W., and Holzer, S. Towards autonomous robotic butlers: Lessons learned with the PR2. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Shangai, China, 2011), ICRA2011, pp. 5568–5575.

[22] Bonasso, R. P., Kortenkamp, D., Miller, D. P., and Slack, M. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence 9* (1995), 237–256.

[23] Borghesan, G., and De Schutter, J. Constraint-based specification of hybrid position-impedance-force tasks. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Hong Kong, 2014), pp. 2290–2296.

[24] Borghesan, G., Willaert, B., De Laet, T., and De Schutter, J. Teleoperation in presence of uncertainties: a constraint-based approach. In *10th IFAC Symposium on Robot Control (SYROCO)* (Dubrovnik, Croatia, September, 5–7 2012), vol. 10.

[25] Brooks, A., Kadous, W., Kaupp, T., Makarenko, A., and Orebäck, A. Orca: Components for robotics. http://orca-robotics.sourceforge.net/, 2004. Last visited January 2015.

[26] Brooks, A., Kaupp, T., Makarenko, A., Orebäck, A., and Williams, S. Towards component based robotics. In *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Edmonton, Canada, 2005), IROS2005, pp. 163–168.

[27] Brooks, A., Kaupp, T., Makarenko, A., Williams, S., and Orebäck, A. Orca: A component model and repository. In *Software Engineering for Experimental Robotics* (2008), Springer Tracts in Advanced Robotics, pp. 231–251.

[28] Brooks, R. A. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation 2*, 1 (1986), 14–23.

[29] Bruyninckx, H. Open robot control software: the OROCOS project. In *Proceedings of the 2001 IEEE International Conference on Robotics and Automation* (Seoul, Korea, 2001), ICRA2001, pp. 2523–2528.

[30] Bruyninckx, H. Robot kinematics and dynamics. Course notes on Advanced Robotics and Control, 2009.

[31] Bruyninckx, H., and De Schutter, J. Specification of force-controlled actions in the "Task Frame Formalism": A survey. *IEEE Transactions on Robotics and Automation 12*, 5 (1996), 581–589.

[32] Bruyninckx, H., Klotzbücher, M., Hochgeschwender, N., Kraetzschmar, G., Gherardi, L., and Brugali, D. The BRICS Component Model: A model-based development paradigm for complex robotics software systems. In *28th ACM Symposium On Applied Computing* (2013), pp. 1758–1764.

[33] Bruyninckx, H., and Soetens, P. Open RObot COntrol Software (OROCOS). http://www.orocos.org/, 2001. Last visited January 2015.

[34] Bruyninckx, H., Soetens, P., and Koninckx, B. The real-time motion control core of the Orocos project. In *Proceedings of the 2003 IEEE International Conference on Robotics and Automation* (Taipeh, Taiwan, 2003), ICRA2003, pp. 2766–2771.

[35] Caccavale, F., Natale, C., Siciliano, B., and Villani, L. Integration for the next generation: embedding force control into industrial robots. *IEEE Robotics and Automation Magazine 12*, 3 (2005), 53–64.

[36] Chiaverini, S., Oriolo, G., and Walker, I. D. Kinematically redundant manipulators. In *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Springer, 2008, pp. 245–268.

[37] Chiaverini, S., and Sciavicco, L. The parallel approach to force/position control manipulators. *IEEE Transactions on Robotics and Automation 9* (1993), 289–293.

[38] DARPA. DARPA grand challenge. http://archive.darpa.mil/grandchallenge04/, 2004. Last visited January 2015.

[39] DARPA. DARPA urban challenge. http://archive.darpa.mil/grandchallenge/, 2004. Last visited January 2015.

[40] DARPA. DARPA grand challenge. http://archive.darpa.mil/grandchallenge05/, 2005. Last visited January 2015.

[41] De Laet, T., and Bellens, S. Geometric semantics software. http://www.orocos.org/wiki/geometric-relations-semantics-wiki, 2012. Last visited January 2015.

[42] De Laet, T., Bellens, S., Bruyninckx, H., and De Schutter, J. Geometric relations between rigid bodies (Part 2): from semantics to software. *IEEE Robotics and Automation Magazine 20*, 2 (2013), 91–102.

[43] De Laet, T., Bellens, S., Smits, R., Aertbeliën, E., Bruyninckx, H., and De Schutter, J. Geometric relations between rigid bodies (Part 1): Semantics for standardization. *IEEE Robotics and Automation Magazine 20*, 1 (2013), 84–93.

[44] De Schutter, J. Improved force control laws for advanced tracking applications. In *Proceedings of the 1988 IEEE International Conference on Robotics and Automation* (Philadelphia, PA, 1988), ICRA88, pp. 1497–1502.

[45] De Schutter, J., Bruyninckx, H., Zhu, W.-H., and Spong, M. W. Force control: a bird's eye view. In *Control Problems in Robotics and Automation: Future Directions*, B. Siciliano, Ed. Springer, San Diego, CA, 1997, pp. 1–17.

[46] De Schutter, J., De Laet, T., Rutgeerts, J., Decré, W., Smits, R., Aertbeliën, E., Claes, K., and Bruyninckx, H. Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty. *The International Journal of Robotics Research 26*, 5 (2007), 433–455.

[47] De Schutter, J., and Leysen, J. Tracking in compliant robot motion: Automatic generation of the task frame trajectory based on observation of the natural constraints. In *Proceedings of the 4th Int. Symposium of Robotics Research* (Santa Cruz, CA, 1987), R. Bolles, Ed., MIT Press.

[48] De Schutter, J., and Van Brussel, H. Compliant Motion I, II. *The International Journal of Robotics Research 7*, 4 (Aug 1988), 3–33.

[49] Decré, W., , Bruyninckx, H., and De Schutter, J. An optimization-based estimation and adaptive control approach for human-robot cooperation. In *12th International Symposium on Experimental Robotics* (Delhi, India, 2010).

[50] Decré, W., , Bruyninckx, H., and De Schutter, J. Extending the Itasc constraint-based robot task specification framework to time-independent trajectories and user-configurable task horizons. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Karlsruhe, Germany, 2013), ICRA2013, pp. 1933–1940.

[51] Decré, W., Smits, R., Bruyninckx, H., and De Schutter, J. Extending iTaSC to support inequality constraints and non-instantaneous task specification. In *Proceedings of the 2009 IEEE International Conference on Robotics and Automation* (Kobe, Japan, 2009), ICRA2009, pp. 964–971.

[52] Dijkstra, E. W. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982, pp. 60–66.

[53] Ding, X. C., Kloetzer, M., Chen, Y., and Belta, C. Automatic deployment of robotic teams. *IEEE Robotics and Automation Magazine 3* (2011), 75–86.

[54] Doherty, P., Heintz, F., and Kvarnström, J. High-level mission specification and planning for collaborative unmanned aircraft systems using delegation. *Unmanned Systems 1*, 1 (2013), 75–119.

[55] Doty, K. L., Melchiorri, C., and Bonivento, C. A theory of generalized inverses applied to robotics. *The International Journal of Robotics Research 12*, 1 (1993), 1–19.

[56] Eclipse. Xtext. http://www.eclipse.org/Xtext/. Last visited January 2015.

[57] Eclipse Foundation. Eclipse Modelling Framework Project. http://www.eclipse.org/modeling/emf/.

[58] Elkady, A., and Sobh, T. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics* (2012).

[59] Eom, K. S., Suh, I. H., Chung, W. K., and Oh, S.-R. Disturbance observer based force control of robot manipulator without force sensor. In *Proceedings of the 1998 IEEE International Conference on Robotics and Automation* (Leuven, Belgium, 1998), ICRA98, pp. 3012–3017.

[60] Escande, A., Mansard, N., and Wieber, P.-B. Fast resolution of hierarchized inverse kinematics with inequality constraints. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Anchorage, Alaska, USA, 2010), pp. 3733–3738.

[61] Estlin, T., Gaines, D., Chouinard, D., Fisher, F., Castano, R., Judd, M., and Nesnas, I. IEEE aerospace conference (iac 05). In *IEEE Aerospace Conference* (Big Sky, MT, USA, 2005).

[62] Finkemeyer, B., Kröger, T., and Wahl, F. M. Executing assembly tasks specified by manipulation primitive nets. *Advanced Robotics 19*, 5 (2005), 591–611.

[63] Finzi, A., Ingrand, F., and Muscettola, N. Model-based executive control through reactive planning for autonomous rovers. In *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Sendai, Japan, 2004), IROS2004, pp. 879–884.

[64] Firby, R. J. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale University, 1989.

[65] Fleury, S., Herrb, M., and Chatila, R. GenoM: a tool for the specification and the implementation of operating modules in a distributed robot architecture. In *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Grenoble, France, 1997), IROS97, pp. 842–848.

[66] Foote, T. Robot Operating System (ROS) geometry stack. http://ros.org/wiki/geometry, 2008. Last visited January 2015.

[67] Ghallab, M., and Laruelle, H. Representation and control in ixtet, a temporal planner. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems* (Chicago, Illinois, USA, 1994), pp. 61–67.

[68] Gregory, N. M., Dorais, G. A., Fry, C., Levinson, R., and Plaunt, C. IDEA: Planning at the core of autonomous reactive agents. In *in Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space* (2002).

[69] Hanafusa, H., Yoshikawa, T., and Nakamura, Y. Analysis and control of articulated robot arms with redundancy. In *Proceedings of the 8th World Congress of the International Federation of Automatic Control* (Kyoto, Japan, 1981), pp. XIV:78–83.

[70] Hogan, N. An organising principle for a class of voluntary movements. *Journal of Neuroscience 4* (1984), 2745–2754.

[71] Hogan, N. Impedance control: An approach to manipulation. Parts I-III. *Transactions of the ASME, Journal of Dynamic Systems, Measurement, and Control 107* (1985), 1–24.

[72] Hogan, N. Stable execution of contact tasks using impedance control. In *Proceedings of the 1987 IEEE International Conference on Robotics and Automation* (Raleigh, NC, 1987), pp. 1047–1054.

[73] Houska, B., Ferreau, H. J., and Diehl, M. ACADO—An open-source toolkit for automatic control and dynamic optimization. In *Proceedings of the 2009 Belgian-French-German Conference on Optimization* (Leuven, Belgium, 2009), p. 167.

[74] Ierusalimschy, R., Celes, W., and de Figueiredo, L. H. Lua Programming Language. http://www.lua.org, 2012. Last visited January 2015.

[75] Inglés-Romero, J. F., Lotz, A., Vicente-Chicote, C., and Schlegel, C. Dealing with run-time variability in service robotics: towards a DSL for non-functional properties. In *3rd International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob)*. Tsukuba, Japan, 2012.

[76] Ingrand, F., and Ghallab, M. Robotics and artificial intelligence: A perspective on deliberation functions. *AI Communications 27*, 1 (2014), 63–80.

[77] Ingrand, F. F., Chatila, R., Alami, R., and Robert, F. PRS: a high level supervision and control language for autonomous mobile robots. In *Proceedings of the 1996 IEEE International Conference on Robotics and Automation* (Minneapolis, MN, 1996), ICRA96, pp. 43–49.

[78] JAUS. JAUS toolset. http://jaustoolset.org/.

[79] JAUS. Reference architecture specification. version 3.3, vol. ii, part 1, the joint architecture for unmanned systems. Tech. rep., Office of the Under Secretary of Defense for Acquisition,Technology and Logistics, Washington DC, USA, 7 2007.

[80] Joyeux, S. ROCK: the RObot Construction Kit. http://www.rock-robotics.org, 2010. Last visited January 2015.

[81] Kanoun, O. Real-time prioritized kinematic control under inequality constraints for redundant manipulators. In *Proceedings of Robotics: Science and Systems* (Los Angeles, USA, June 2011).

[82] Kanoun, O., Lamiraux, F., and Wieber, P.-B. Kinematic control of redundant manipulators: Generalizing the task-priority framework to inequality task. *IEEE Transactions on Robotics 27*, 4 (2011), 785–792.

[83] Kanoun, O., Lamiraux, F., Wieber, P.-B., Kanehiro, F., Yoshida, E., and Laumond, J.-P. Prioritizing linear equality and inequality systems: Application to local motion planning for redundant robots. In *Proceedings of the 2009 IEEE International Conference on Robotics and Automation* (Kobe, Japan, 2009), ICRA2009, pp. 724–729.

[84] Katsura, S., Matsumoto, Y., and Ohnishi, K. Modeling of force sensing and validation of disturbance observer for force control. *IEEE Transactions on Industrial Electronics 54*, 1 (feb. 2007), 530 –538.

[85] Kazerooni, H., Sheridan, T. B., and Houpt, P. K. Robust compliant motion for manipulators, Part I: The fundamental concepts of compliant motion. *IEEE Journal of Robotics and Automation RA-2* (1986), 83–92.

[86] Khatib, O. A unified approach for motion and force control of robot manipulators: The operational space formulation. *IEEE Journal of Robotics and Automation RA-3*, 1 (1987), 43–53.

[87] Klein, C. A., and Blaho, B. Dexterity measures for the design and control of kinematically redundant manipulators. *The International Journal of Robotics Research 6*, 2 (1987), 72–83.

[88] Klotzbücher, M. *Domain specific languages for hard real-time safe coordination of robot and machine tool systems.* PhD thesis, KU Leuven, Department of Mechanical Engineering, 2013.

[89] Klotzbücher, M., Biggs, G., and Bruyninckx, H. Pure coordination using the coordinator–configurator pattern. In *Proceedings of the 3rd International Workshop on Domain-Specific Languages and models for ROBotic systems* (November 2012).

[90] Klotzbücher, M., and Bruyninckx, H. Hard real-time control and coordination of robot tasks using Lua. In *Proceedings of the Thirteenth Real-Time Linux Workshop* (October 2011).

[91] Klotzbücher, M., and Bruyninckx, H. Coordinating robotic tasks and systems with rFSM Statecharts. *Journal of Software Engineering in Robotics 3*, 1 (2012), 28–56.

[92] Klotzbücher, M., and Bruyninckx, H. A lightweight, composable metamodelling language for specification and validation of internal domain specific languages. In *Proceedings of the 8th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, vol. 7706 of *Springer Lecture Notes in Computer Science*. 2012, pp. 58–68.

[93] Klotzbücher, M., Smits, R., Bruyninckx, H., and De Schutter, J. Reusable hybrid force-velocity controlled motion specifications with executable domain specific languages. In *Proceedings of the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems* (San Francisco, California, 2011), IROS2011, pp. 4684–4689.

[94] Klotzbücher, M., Soetens, P., and Bruyninckx, H. OROCOS RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages. In *International Workshop on Dynamic languages for RObotic and Sensors* (2010), pp. 284–289.

[95] Kortenkamp, D., and Simmons, R. G. Robotic systems architectures and programming. In *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Springer, 2008, pp. 187–206.

[96] Kress-Gazit, H., Wongpiromsarn, T., and Topcu, U. Correct, reactive, high-level robot control. *IEEE Robotics and Automation Magazine 3* (2011), 65 –74.

[97] Kröger, T., and Finkemeyer, B. Robot motion control during abrupt switchings between manipulation primitives. In *Workshop on Mobile Manipulation at the IEEE International Conference on Robotics and Automation* (Shangai, China, May 2011).

[98] Kröger, T., Finkemeyer, B., Heuck, M., and Wahl, F. M. Compliant motion programming: The Task Frame Formalism revisited. *Journal of Robotics and Mechatronics 3* (2004), 1029–1034.

[99] KUKA YouBot. Youbot ros packages. https://github.com/youbot/youbot-ros-pkg/. Accessed online 1 January 2015.

[100] Maciejewski, A. A., and Klein, C. A. Obstacle avoidance for kinematically redundant manipulators in dynamically varying environments. *The International Journal of Robotics Research 4*, 3 (1985), 109–117.

[101] Maciejewski, A. A., and Klein, C. A. Numerical filtering for the operation of robotic manipulators through kinematically singular configuration. *Journal of Robotic Systems 5*, 6 (1988), 527–552.

[102] Mansard, N., Stasse, O., Evrard, P., and Kheddar, A. A versatile generalized inverted kinematics implementation for collaborative working humanoid robots: The Stack of Tasks. In *Proceedings of the 2009 International Conference on Advanced Robotics* (Munich, Germany, 2009), ICAR2009, pp. 1–6.

[103] Martin, R. C. The dependency inversion principle. 1996, pp. 1–12. http://www.objectmentor.com/resources/articles/dip.pdf.

[104] Mason, M. T. Compliance and force control for computer controlled manipulators. *IEEE Transactions on Systems, Man, and Cybernetics SMC-11*, 6 (1981), 418–432.

[105] McGann, C., Py, F., Rajan, K., Ryan, J. P., and Henthorn, R. Adaptive Control for Autonomous Underwater Vehicles. In *Proceedings of the AAAI National Conference on Artificial Intelligence* (Chicago, Illinois, USA, 2008).

[106] McGann, C., Py, F., Tajan, K., Thomas, H., Henthorn, R., and McEwen, R. A deliberative architecture for AUV control. In *Proceedings of the 2008 IEEE International Conference on Robotics and Automation* (Pasadena, California, U.S.A., 2008), pp. 1049–1054.

[107] Mernik, M., Heering, J., and Sloane, A. M. When and how to develop domain-specific languages. *ACM Comput. Surv. 37*, 4 (Dec. 2005), 316–344.

[108] Michal, D. S., and Etzkorn, L. A comparison of player/stage/gazebo and microsoft robotics developer studio. In *Proceedings of the 49th Annual Southeast Regional Conference* (2011), ACM-SE '11, ACM Press, pp. 60–66.

[109] Miller, J., and Mukerji, J. MDA Guide version 1.0.1. Tech. rep., Object Management Group (OMG), 2003.

[110] Nakamura, Y. *Advanced robotics: redundancy and optimization.* Addison-Wesley, Reading, MA, 1991.

[111] Nakamura, Y., and Hanafusa, H. Inverse kinematic solutions with singularity robustness for robot manipulator control. *ASMEDSMC 108* (1986), 163–171.

[112] Nakamura, Y., Hanafusa, H., and Yoshikawa, T. Task-priority based redundancy control of robot manipulators. *The International Journal of Robotics Research 6*, 2 (1987), 3–15.

[113] Natale, C. *Interaction control of robot manipulators—Six-Degrees-of-Freedom tasks*, vol. 3 of *Springer Tracts in Advanced Robotics.* Springer-Verlag, London, UK, 2003.

[114] National Institute of Advanced Industrial Science and Technology, Intelligent Systems Research Institute. OpenRTM-Aist. http://www.openrtm.org. Last visited January 2015.

[115] Nesnas, I. A. D., Simmons, R., Gaines, D., Kunz, C., Diaz-Calderon, A., Estlin, T., Madison, R., Guineau, J., McHenry, M., Shu, I.-H., and Apfelbaum, D. CLARAty: Challenges and steps toward reusable robotic software. *International Journal of Advanced Robotic Systems 3*, 1 (2006), 23–30.

[116] Nguyen, H., Ciocarlie, M., Hsiao, K., and Kemp, C. ROS Commander (ROSCo): Behavior creation for home robots. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Karlsruhe, Germany, 2013), ICRA2013, pp. 467–474.

[117] NI. LabVIEW. http://www.ni.com/labview/.

[118] Nilsson, N. J. A mobile automaton: an application of AI techniques. In *Proceedings of the First International Joint Conference on Artificial Intelligence* (Washington, D. C., 1969), pp. 509–520.

[119] Nilsson, N. J. Hierarchical robot planning and execution system. Tech. rep., Stanford Research Institute, 1973.

[120] Nilsson, N. J. *Principles of artificial intelligence*. Tioga Publishing Co., Palo Alto, CA, 1980.

[121] Nordmann, A., and Wrede, S. A domain-specific language for rich motor skill architectures. In *3rd International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob)*. Tsukuba, Japan, 2012.

[122] Object Management Group. OMG. http://www.omg.org.

[123] Object Management Group. Robotic Technology Component. http://www.omg.org/spec/RTC/. Accessed online January 2015.

[124] Object Management Group. Unified Modeling Language (UML) superstructure specification, version 2.4.1. http://www.uml.org/, 2011.

[125] Open Management Group. CORBA: Component Model. http://ditec.um.es/~dsevilla/ccm//.

[126] Penrose, R. A generalized inverse for matrices. *Proceedings of the Cambridge Philosophical Society 51*, 3 (1955), 406–413.

[127] Philips, J. *Holonic task execution control of multi-mobile-robot systems*. PhD thesis, KU Leuven, Department of Mechanical Engineering, 2012.

[128] Prassler, E., Bruyninckx, H., Nilsson, K., and Shakhimardanov, A. The use of reuse for designing and manufacturing robots. Tech. rep., Robot Standards project, 2009. http://www.robot-standards.eu/Documents_RoSta_wiki/whitepaper_reuse.pdf.

[129] Py, F., and Ingrand, F. Real-time execution control for autonomous systems. In *Embedded and Real-Time Systems* (2004), pp. 21–23.

[130] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T. B., Leibs, J., Wheeler, R., and Ng, A. Y. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software* (2009).

[131] Radestock, M., and Eisenbach, S. Coordination in evolving systems. In *Trends in Distributed Systems. CORBA and Beyond*. Springer-Verlag, 1996, pp. 162–176.

[132] Raibert, M., and Craig, J. J. Hybrid position/force control of manipulators. *Transactions of the ASME, Journal of Dynamic Systems, Measurement, and Control 102* (1981), 126–133.

[133] RoboHow. The RoboHow Project. http://robohow.eu/.

[134] Rutgeerts, J. *Constraint-based task specification and estimation for sensor-based robot tasks in the presence of geometric uncertainty*. PhD thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, 2007.

[135] Saab, L., Ramos, O. E., Keith, F., Mansard, N., Souères, P., and Fourquet, J.-Y. Dynamic whole-body motion generation under rigid contacts and other unilateral constraints. *IEEE Transactions on Robotics 29*, 2 (2013), 346–362.

[136] Salisbury, J. K. Active stiffness control of a manipulator in Cartesian coordinates. In *19th IEEE Conf. on Decision and Control* (1980).

[137] Samson, C., Le Borgne, M., and Espiau, B. *Robot Control, the Task Function Approach*. Clarendon Press, Oxford, England, 1991.

[138] Schreiber, G., Stemmer, A., and Bischoff, R. The Fast Research Interface for the KUKA Lightweight Robot. In IEEE Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications – How to Modify and Enhance Commercial Controllers (ICRA 2010), May 2010.

[139] Sentis, L., and Khatib, O. Synthesis of whole-body behaviors through hierarchical conrol of behavioral primitives. *International Journal of Humanoid Robotics 2*, 4 (2005), 505–518.

[140] Sentis, L., and Khatib, O. A whole-body control framework for humanoid operating in human environments. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation* (Orlando, U.S.A., 2006), ICRA2006, pp. 2641–2648.

[141] Siciliano, B., and Khatib, O. E. *Springer Handbook of Robotics*. Springer-Verlag, Berlin, Heidelberg, 2008.

[142] Siciliano, B., and Slotine, J.-J. E. A general framework for managing multiple tasks in highly redundant robotic systems. In *Proceedings of the 1991 International Conference on Advanced Robotics* (Pisa, Italy, 1991), pp. 1211–1216.

[143] Simmons, R., and Apfelbaum, D. A task description language for robot control. In *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Vancouver, British Columbia, Canada, 1998), IROS98, pp. 1931–1937.

[144] Smits, R. *Robot skills: design of a constraint-based methodology and software support*. PhD thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, May 2010.

[145] Smits, R., and Bruyninckx, H. Composition of complex robot applications via data flow integration. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Shangai, China, 2011), ICRA2011, pp. 5576–5580.

[146] Smits, R., Bruyninckx, H., and Aertbeliën, E. KDL: Kinematics and Dynamics Library. http://www.orocos.org/kdl, 2001. Last visited January 2015.

[147] Smits, R., Bruyninckx, H., and De Schutter, J. Software support for high-level specification, execution and estimation of event-driven, constraint-based multi-sensor robot tasks. In *Proceedings of the 2009 International Conference on Advanced Robotics* (Munich, Germany, 2009), ICAR2009.

[148] Smits, R., De Laet, T., Claes, K., Bruyninckx, H., and De Schutter, J. iTaSC: a tool for multi-sensor integration in robot manipulation. In *IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems* (Seoul, South-Korea, 2008), MFI2008, pp. 426–433.

[149] Soetens, P. *A Software Framework for Real-Time and Distributed Robot and Machine Control*. PhD thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, May 2006. http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf.

[150] Software Engineering Institute, Carnegie Mellon University. Glossary of software architecture terms. http://www.sei.cmu.edu/architecture/start/glossary/index.cfm. Accessed online 10 January 2015.

[151] Tachi, S., Sakaki, T., Arai, H., Nishizawa, S., and Pelaez-Polo, J. F. Impedance control of a direct-drive manipulator without using force sensors. *Advanced Robotics* (1991), 183–205.

[152] Taylor, R. N., Medvidovic, N., and Dashofy, E. M. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[153] The MathWorks. Design and simulate state charts by The Mathworks. www.mathworks.de/products/stateflow/.

[154] The MathWorks. Simulation and model-based design by The MathWorks. http://www.mathworks.com/products/simulink/.

[155] Thomas, U., Finkemeyer, B., Kröger, T., and Wahl, F. M. Error-tolerant execution of complex robot tasks based on skill primitives. In *Proceedings of the 2003 IEEE International Conference on Robotics and Automation* (Taipeh, Taiwan, 2003), ICRA2003, pp. 3069–3075.

[156] Thomas, U., Hirzinger, G., Rumpe, B., Schulze, C., and Wortmann, A. A new skill based robot programming language using uml/p statecharts. In *Proceedings of the IEEE International Conference on Robotics and Automation* (Karlsruhe, Germany, 2013), ICRA2013, pp. 461–466.

[157] Tsang, E. P. K. *Foundations of constraint satisfaction*. Computation in cognitive science. Academic Press, 1993.

[158] van de Molengraft, M. RoboEarth. http://www.roboearth.org/, 2011.

[159] Van de Poel, P., Witvrouw, W., Bruyninckx, H., and De Schutter, J. An environment for developing and optimizing compliant robot motion tasks. In *Proceedings of the 1993 International Conference on Advanced Robotics* (Tokyo, Japan, 1993), pp. 713–718.

[160] van Deursen, A., Klint, P., and Visser, J. Domain-specific languages: an annotated bibliography. *SIGPLAN Not. 35*, 6 (June 2000), 26–36.

[161] Vanthienen, D. Prioritized, weighted, damped least-squares solver. http://gitlab.mech.kuleuven.be/rob-itasc/itasc_solvers.git, 2011. Last visited January 2015.

[162] Vanthienen, D. iTaSC tutorials. http://orocos.org/wiki/orocos/itasc-wiki/itasc-tutorials, 2013. Last visited January 2015.

[163] Vanthienen, D. Composition pattern for constraint-based programming with application to force-sensorless robot tasks: Online overview of released code. http://people.mech.kuleuven.be/~dvanthienen/thesis/code.html, 2015. Last visited January 2015.

[164] Vanthienen, D. Composition pattern for constraint-based programming with application to force-sensorless robot tasks: Online videos of experiments. http://people.mech.kuleuven.be/~dvanthienen/thesis/videos.html, 2015. Last visited January 2015.

[165] Vanthienen, D., De Laet, T., Decré, W., Bruyninckx, H., and De Schutter, J. Force-sensorless and bimanual human-robot comanipulation. In *10th IFAC Symposium on Robot Control (SYROCO)* (Dubrovnik, Croatia, September, 5–7 2012), vol. 10.

[166] Vanthienen, D., De Laet, T., Smits, R., and Bruyninckx, H. itasc software. http://www.orocos.org/itasc, 2011. Last visited January 2015.

[167] Vanthienen, D., Klotzbücher, M., De Laet, T., De Schutter, J., and Bruyninckx, H. Rapid application development of constrained-based task modelling and execution using domain specific languages. In *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Tokyo, Japan, 2013), IROS2013, pp. 1860–1866.

[168] Villani, L., and De Schutter, J. Force control. In *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Springer, 2008, pp. 161–185.

[169] Wampler, C. W. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares solutions. *IEEE Transactions on Systems, Man, and Cybernetics 16* (1986), 93–101.

[170] Whitney, D. E. Force feedback control of manipulator fine motions. *Transactions of the ASME, Journal of Dynamic Systems, Measurement, and Control 99*, 2 (1977), 91–97.

[171] Willow Garage. Robot Operating System (ROS). http://www.ros.org, 2008. Last visited January 2015.

[172] Willow Garage. Universal Robot Description Format (URDF). http://www.ros.org/wiki/urdf, 2009.

[173] Willow Garage. Willow Garage. http://www.willowgarage.com/, 2011. Last visited January 2015.

[174] Witvrouw, W., Najéra, J., De Schutter, J., and Laugier, C. Integrating assembly planning with compliant control. In *Proceedings of the World Automation Congress 1996* (Montpellier, France, 1996), vol. 6, pp. 523–528.

[175] Witvrouw, W., Van de Poel, P., and De Schutter, J. Comrade: Compliant motion research and development environment. In *3rd IFAC/IFIP workshop on Algorithms and Architectures for Real-Time Control* (Ostend, Belgium, 1995), pp. 81–87.

[176] Yoshikawa, T. Force control of robot manipulators. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation* (San Francisco, CA, 2000), ICRA2000, pp. 220–226.

List of Publications

Articles in Internationally reviewed Journals

- Vanthienen, D., De Laet, T., Bruyninckx, H. Systematic robot application development: applying the Composition Pattern to constraint-based programming. *Submitted to Robotics and Automation Magazine.*
- Vanthienen, D., Klotzbücher, M., Bruyninckx, H. The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming. *Journal of Software Engineering for Robotics (JOSER)*, 5 (1), pp. 17-35.

Papers at International Conferences and Symposia, Published in Full in Proceedings

- Vanthienen, D., Klotzbücher, M., De Laet, T., De Schutter, J., and Bruyninckx, H. Rapid application development of constrained-based task modelling and execution using domain specific languages. In *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Tokyo, Japan, 2013), IROS2013, pp. 1860–1866.
- Vanthienen, D., De Laet, T., Decré, W., Bruyninckx, H., and De Schutter, J. Force-sensorless and bimanual human-robot comanipulation. In *10th IFAC Symposium on Robot Control (SYROCO)* (Dubrovnik, Croatia, September, 5–7 2012), vol. 10.

Meeting Abstracts, Presented at International Conferences and Symposia, Published or not Published in Proceedings or Journals

- Vanthienen, D., De Laet, T., De Schutter, J., Bruyninckx, H. (2013). Software framework for robot application development: a constraint-based task programming approach. *IEEE International Conference on Robotics and Automation (ICRA) SDIR workshop.* Karlsruhe, 6 May 2013.

- Vanthienen, D., Robyns, S., Aertbeliën, E., De Schutter, J. (2013). Force-sensorless robot force control within the instantaneous task specification and estimation (iTaSC) framework. *Benelux Meeting on Systems and Control.* Houffalize, Belgium, 26-28 March 2013.

- Vanthienen, D., De Laet, T., Decré, W., De Schutter, J. (2013). Acceleration- vs. velocity-resolved constraint-based instantaneous task specification and estimation (iTaSC). *Benelux Meeting on Systems and Control 2013.* Houffalize, Belgium, 26-28 March 2013.

- Vanthienen, D., De Laet, T., Decré, W., Smits, R., Klotzbücher, M., Buys, K., Bellens, S., Gherardi, L., Bruyninckx, H., De Schutter, J. (2011). iTaSC as a unified framework for task specification, control, and coordination, demonstrated on the PR2. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* San Francisco, 25-30 September 2011.

- Buys, K., Bellens, S., Vanthienen, D., Decré, W., Klotzbücher, M., De Laet, T., Smits, R., Bruyninckx, H., De Schutter, J. (2011). Haptic coupling with the PR2 as a demo of the OROCOS - ROS - Blender integration. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* San Francisco, California, 25-30 September 2011.

- Buys, K., Bellens, S., Vanthienen, D., De Laet, T., Smits, R., Klotzbücher, M., Decré, W., Bruyninckx, H., De Schutter, J. (2011). Haptic coupling with augmented feedback between the KUKA youBot and the PR2 robot arms. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* San Francisco, California, 25-30 September, 2011.

- Vanthienen, D., De Laet, T., Smits, R., Buys, K., Bellens, S., Klotzbücher, M., Bruyninckx, H., De Schutter, J. (2011).

Demonstration of iTaSC as a unified framework for task specification, control, and coordination for mobile manipulation. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* San Francisco, 25-30 September 2011.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF MECHANICAL ENGINEERING
PRODUCTION ENGINEERING, MACHINE DESIGN AND AUTOMATION DIVISION
Celestijnenlaan 300B box 2420
B-3001 Heverlee
info@mech.kuleuven.be
http://www.mech.kuleuven.be